
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**
**BRNO UNIVERSITY OF TECHNOLOGY**

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**
**FACULTY OF INFORMATION TECHNOLOGY**

**ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ**
**DEPARTMENT OF COMPUTER SYSTEMS**


# SPECIFIKACE SCÉNÁŘŮ PORTOVATELNÝCH STIMULŮ PRO MODULY PROCESORU RISC-V

**PORTABLE STIMULUS SCENARIOS SPECIFICATION FOR RISC-V PROCESSOR MODULES**

**DIPLOMOVÁ PRÁCE**
**MASTER'S THESIS**


**AUTOR PRÁCE** **Bc. PETR BARDONEK**
**AUTHOR**

**VEDOUCÍ PRÁCE** **Ing. MARCELA ZACHARIÁŠOVÁ, Ph.D.**
**SUPERVISOR**



**BRNO 2018**

**Vysoké učení technické v Brně - Fakulta informačních technologií**

Ústav počítačových systémů Akademický rok 2017/2018

# Zadání diplomové práce

Řešitel: **Bardonek Petr, Bc.**
Obor: Počítačové a vestavěné systémy
Téma: **Specifikace scénářů portovatelných stimulů pro moduly procesoru RISC-V**
**Portable Stimulus Scenarios Specification for RISC-V Processor Modules**
Kategorie: Počítačová architektura

Pokyny:

1. Seznamte se s problematikou funkční verifikace číslicových systémů a s novým standardem pro Portable Stimulus.
2. Seznamte se s verifikační platformou Questa od společnosti Mentor a s nástrojem InFact této platformy.
3. Seznamte se s vybraným modulem/moduly procesoru RISC-V.
4. Implementujte scénáře portovatelných stimulů pro vybraný modul/moduly procesoru RISC-V v nástroji InFact.
5. Otestujte výstupy nástroje InFact ve verifikačním prostředí běžícím v simulačním nástroji Questa.
6. Vyhodnoťte úroveň portovatelnosti scénářů pro moduly procesoru na scénáře pro top-level úroveň procesoru.

Literatura:

- RISC-V Foundation, 2016. User-level RISC-V ISA specification, https://riscv.org/specifications/.
- RISC-V Foundation, 2017. Draft Privileged RISC-V ISA specification, https://riscv.org/specifications/.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Zachariášová Marcela, Ing., Ph.D.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

prof. Ing. Lukáš Sekanina, Ph.D.
*vedoucí ústavu*

# Abstrakt

Práce se zabývá návrhem a implementací verifikačních scénářů portovatelných stimulů pro vybrané moduly procesoru Berkelium implementujícím architekturu RISC-V od společnosti Codasip. Cílem této práce je s využitím nového standardu pro Portable Stimulus připravovaného organizací Accellera navrhnout a implementovat scénáře portovatelných stimulů za použití nástroje Questa InFact od společnosti Mentor. Takto navržené scénáře portovatelných stimulů se připojí k již existujícím verifikačním prostředím vytvořených podle metodiky UVM a následně se pomocí nich provede verifikace modulů procesoru Berkelium implementujícím architekturu RISC-V. Poslední částí práce je vyhodnocení úrovně portovatelnosti implementovaných scénářů do jednotlivých úrovní procesoru Berkelium implementujícím architekturu RISC-V (IP bloky, subsystémy, systémy jako celek), kdy je snahou využít navržené scénáře napříč všemi verifikovanými úrovněmi.



# Abstract

The thesis is focused on the design and implementation of the portable stimulus verification scenarios for selected Berkelium processor modules based on RISC-V architecture from Codasip. The aim of this work is to use new standard for Portable Stimulus developed by Accellera organization to design and implement portable stimulus scenarios using the Questa InFact tool from Mentor. The proposed portable stimulus scenarios are then linked to the already existing verification environments of the UVM methodology and then they are used for verification of the Berkelium processor modules based on RISC-V architecture. The last part of the thesis is the evaluation of portability of the implemented scenarios to the individual levels of the Berkelium processor based on RISC-V architecture (IP blocks, subsystems, system level), in which it tries to use the proposed scenarios across all verificated levels.



# Klíčová slova

funkční verifikace, RISC-V, Portovatelné stimuly, procesory Berkelium, Questa InFact

# Keywords

functional verification, RISC-V, Portable Stimulus, Berkelium processors, Questa InFact


# Citace

BARDONEK, Petr. *Specifikace scénářů portovatelných stimulů pro moduly procesoru RISC-V*. Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Marcela Zachariášová, Ph.D.

# Specifikace scénářů portovatelných stimulů pro moduly procesoru RISC-V

## Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením paní Ing. Marcely Zachariášové Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Petr Bardonek
30. května 2018


## Poděkování

Především chci poděkovat vedoucí mé diplomové práce paní Ing. Marcele Zachariášové Ph.D. za odborné vedení, ochotu při konzultacích a morální podporu. Dále chci poděkovat za rady a morální podporu zaměstnancům společnosti Codasip.

# Obsah

# Kapitola 1

# Úvod

Počítačové a vestavěné systémy se staly nedílnou součástí každodenního života mnoha z nás. Pro tyto systémy je důležité, aby pracovaly bezchybně podle zadané specifikace. Jsou na ně kladeny stále vyšší nároky a s nimi stoupá i jejich složitost. Neustále narůstající složitost činí velký problém při implementaci požadovaných funkcí systému tak, jak byly navrženy při specifikaci bez toho, aby byly do systému vneseny chyby. Obor zabývající se touto problematikou se nazývá *verifikace*. Způsobů verifikace systémů je mnoho, ale v této práci se budeme zabývat pouze v současné době nejrozšířenějším typem verifikace v této oblasti a to *funkční verifikací*.

Mezi počítačové a vestavěné systémy patří také v poslední době jeden z nejvíce se rozšiřujících systémů tzv. *systém na čipu*, dále už jen SoC (angl. *System on a Chip*). S narůstající komplexností SoC je třeba zvýšit produktivitu jejich verifikace. Jedním z problémů se stává samotná verifikace, neboť pro verifikaci SoC je nutné použít více verifikačních přístupů. Využívá se jak simulace, tak emulace a FPGA prototypování, přičemž se soustředíme postupně na různé úrovně systému (IP bloky, subsystémy, systém jako celek). Pro každý přístup je definice verifikačních scénářů prováděna jiným způsobem a jiným programovacím jazykem, a proto použití verifikačních testů napříč všemi verifikačními přístupy(úrovněmi verifikovaného systému) je časově velice náročné a neobejde se bez chyb. Dále pak na systémové úrovni je vytváření verifikačních testů velmi obtížné a většinou se jedná pouze o verifikační testy přímé (typy testů a jejich popis je rozebrán v 2.2.2), což vede ke snížení efektivnosti verifikace systému (je výhodnější použití pseudonáhodně generovaných verifikačních testů). Rovněž také nedochází k verifikaci okrajových případů, protože interakce na systémové úrovni jsou tak komplexní, že dosáhnout jich pouze za pomoci verifikačních testů přímých je téměř nereálné[17].

Nový Portable Stimulus Standard od organizace Accellera se pokouší odstranit tyto problémy zavedením jednotné specifikace stimulů a verifikačních scénářů (stimuly jsou data, která aplikujeme na vstupy verifikovaného obvodu). Pomocí tohoto standardu budou schopni uživatelé specifikovat sadu verifikačních scénářů, vygenerovat jejich implementace v různých programovacích jazycích a následně je využít napříč všemi verifikačními přístupy(úrovněmi verifikovaného systému), pomocí kterých je potřeba provést verifikaci [2][4]. Dalším vylepšením, které má nový standard přinést je možnost vytvořit komplexní verifikační testy i na systémové úrovni [5], kde je doposud možné vytvářet většinou pouze verifikační testy přímé, což je velkým problémem pro dosažení okrajových případů testovaného systému.

Klíčem k portovatelným stimulům je zvýšení úrovně abstrakce nad úroveň tzv. transakcí [17] (pojem transakce je vysvětlen v kapitole 2.4.1). Při použití pseudonáhodných verifikač-

ních testů pro pokrytí požadovaného chování nelze předem určit počet transakcí k tomu potřebných, načež dochází k vysoké redundanci vstupních transakcí a zvýšení časové náročnosti. Namísto jednotlivých sekvencí nový standard definuje možnost vytvořit relevantní scénáře, na které je potřeba se při verifikaci zaměřit a tím výrazné snížit redundanci generovaných transakcí při ověřování okrajových případů. K tomu se využívá grafové reprezentace, ve které jsou definovány kritické akce a jsou zavedena různá omezení umožňující definovat více scénářů z jediné specifikace.

Cílem této práce je za pomoci nástroje Questa InFact od společnosti Mentor navrhnout a vytvořit portovatelné verifikační scénáře pro moduly procesoru Berkelium implementujícím architekturu RISC-V od společnosti Codasip s využitím již vytvořených verifikačních prostředí. Verifikační prostředí vytvořená společností Codasip využívají metodiku UVM (angl. *Universal Verification Methodology*), která je popsána v kapitole 2. Nástroj Questa InFact nám dává možnost vygenerovat implementaci pro vytvořené portovatelné verifikační scénáře jako samostatnou komponentu UVM metodiky a připojit ji již k dříve vytvořenému verifikačnímu prostředí. Samotné připojení takto vytvořené komponenty vyžaduje pouze malé změny v daném verifikačním prostředí [5]. Dalším cílem této práce je vzít takto vytvořené verifikační scénáře pro jednotlivé moduly, použít je při verifikaci na vyšších úrovních procesoru Berkelium a dostat se tak až k jejich opětovnému použití při verifikaci na systémové úrovni.

Text práce je rozdělen do několika kapitol. V následující kapitole jsou popsány a vysvětleny základy funkční verifikace a metodika UVM. Nástroj Questa InFact od společnosti Mentor ve kterém probíhá implementace portovatelných verifikačních scénářů je popsán v kapitole 3. V kapitole 4 je popisován procesor Berkelium implementující architekturu RISC-V od společnosti Codasip spolu s jeho moduly, pro které byly navrženy a následně implementovány portovatelné verifikační scénáře.

# Kapitola 2

# Funkční verifikace

Po každém počítačovém systému chceme, aby prováděl bezchybně svou funkci, jež je uvedena na začátku jeho návrhu v specifikaci. Úkolem verifikace je odhalit co nejvíce chyb systému v počátečních fázích verifikace (v simulaci) a pokud možno se vyhnout nalezení chyb až na vyrobeném skutečném hardwaru, protože jeho výroba je drahá. To je také jeden z hlavních důvodů vytvoření oboru verifikace. Úkolem verifikačního inženýra je odhalit co největší množství chyb a zajistit, že finální produkt bude pracovat jak bylo zamýšleno při specifikaci [26].

Funkční verifikace se řadí mezi techniky založené na simulaci s tím rozdílem, že využívá řadu přídavných technik pro zvýšení efektivity, které činí její verifikační prostředí (angl. *verification environment, HVL testbench*) mnohem komplexnější. Funkční verifikace je implementována v jazyce SystemVerilog určeném pro verifikaci (angl. *Hardware Verification Language*, HVL), který přináší řadu dalších vylepšení. Obvod určený k verifikaci (v dalším textu budeme využívat pro verifikovaný obvod zkratku DUT z angl. *Design Under Test*) je implementován v jednom z jazyků určených pro modelování hardwaru (angl. *Hardware Description Language*, HDL).

Na následujících stránkách této kapitoly jsou nejdříve stručně popsány základy verifikace založené na simulaci, pak jsou postupně vysvětleny přídavné techniky zvyšující efektivitu funkční verifikace, jsou zmíněny verifikační výzvy, je popsán jazyk SystemVerilog, který se stal nejpopulárnějším z řady HVL jazyků, je blíže vysvětlena jedna z metodik, které jsou definovány pro usnadnění tvorby znovupoužitelných a snadno rozšiřitelných verifikačních prostředí a je popsán celý verifikační cyklus podle [26].

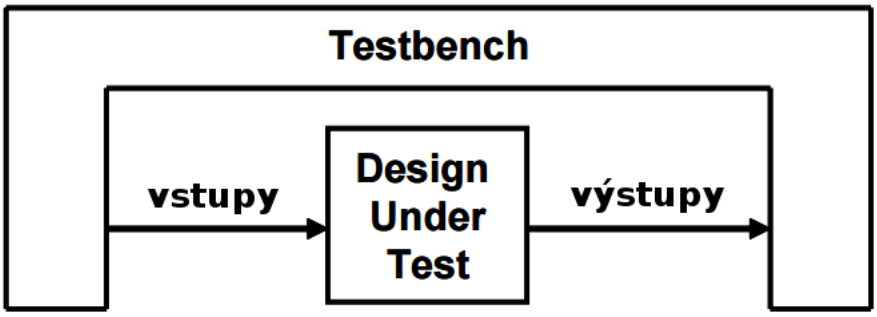
## 2.1 Verifikace založená na simulaci

Základním principem tohoto typu verifikace je vytvoření tzv. *verifikačního prostředí* (angl. *verification environment, HVL testbench*), které slouží k simulování prostředí do nějž bude DUT zasazeno (viz obrázek 2.1). Cílem verifikačního prostředí je ověřit zda DUT plní správně funkce, které jsou uvedeny ve specifikaci. Navzdory tomu, že verifikace založená na simulaci je nejčastěji používaný přístup v praxi, nezaručuje na 100%, že v systému se už nevyskytují další chyby na rozdíl například od formální verifikace, která je pro změnu náročná na použití.

Základními kroky jsou:

- Vygenerování stimulů a jejich následné přivedení na vstupy DUT.
- Automatické zachycení a kontrola správnosti výstupů.

- Měření pokroku verifikace pomocí vhodně zvolených metrik pro detekci jejího ukončení.




Obrázek 2.1: Základní schéma zapojení DUT a testbenche [23]

Výhodou verifikací založených na simulaci je jednoduchost implementace, nenáročnost na pochopení, možnost reprodukce chyb skrze opakované pouštění testovacího scénáře, a také velká podpora ze strany simulačních nástrojů, které umožňují zpětně projít celý průběh simulace. Problém je ovšem s časovou náročností takové verifikace, která je přímo úměrná komplexnosti DUT. Simulace je výpočetně náročná úloha a pokud bychom chtěli verifikovat všechny možné stavy DUT, trvala by verifikace i jednodušších hardwarových obvodů celé dny.

## 2.2 Přídavné techniky zvyšující efektivitu

Tato podkapitola pojednává o technikách zavedených funkční verifikací. Tyto techniky výrazně zlepšují efektivitu verifikace založené na simulaci a odlišují tak funkční verifikaci od jednoduššího typu testování (např. pomocí HDL testbenchů).

### 2.2.1 Verifikace řízená pokrytím

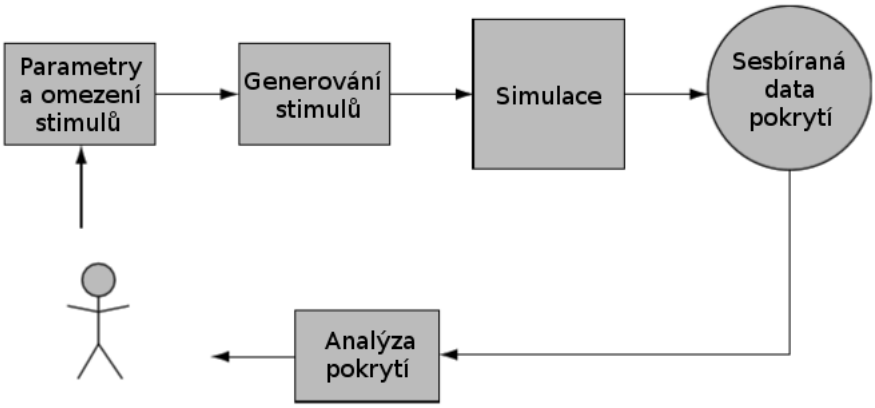
Jedním z nejtěžších úkolů je určit, kdy funkční verifikace dosahuje takového stavu, že je možno ji ukončit. K tomu slouží metriky pokrytí. Je dobrým zvykem používat výsledky měření daných metrik po provedení simulace k úpravě, či dokonce vytvoření dalších verifikačních testů pro dosažení co nejvyššího (nejlépe 100%) pokrytí. Celý tento proces je zobrazen na obrázku 2.2.

Dosažení 100% pokrytí nezaručuje absenci chyb, stále se jedná o verifikaci založenou na simulaci a můžeme tedy ručit pouze za tu funkcionalitu, jenž byla verifikována.

#### Strukturní pokrytí

Jedná se o pokrytí samotné implementace DUT, kdy je měřeno množství kódu vykonaného během simulace. Zahrnuje v sobě hned několik metrik pokrytí pro typické strukturní modely [26]:

- **pokrytí přepínání** (angl. *toggle coverage*) - měří kolikrát signály v HDL modelu změnily svou logickou hodnotu během simulace, absence změn signálů v nějaké oblasti DUT značí, že stimuly se na tuto oblast vůbec nezaměřily.




Obrázek 2.2: Úprava stimulů na základě analýzy metrik [26]

- **pokrytí výrazů** (angl. *statement coverage*) nebo také **pokrytí řádků** (angl. *line coverage*) - bere syntaktickou strukturu HDL specifikace a měří, které řádky byly vykonány během simulačního běhu.
- **pokrytí větvení** (angl. *branch coverage*) nebo také **pokrytí podmínek** (angl. *conditional coverage*) - sleduje podmíněné stavy v HDL a uchovává záznam o tom, které podmínky při simulaci nastaly a které ne.
- **pokrytí cest** (angl. *path coverage*) - jedná se o zjemnění branch coverage, kdy místo, aby tato metrika sledovala jediné podmíněné rozhodnutí v izolaci, provádí analýzu provedeného průchodu HDL a identifikuje kombinace po sobě jdoucích rozhodnutí.
- **pokrytí konečného stavového automatu** (angl. *finite state machine coverage*, FSM) - zachycuje stavy, které byly navštíveny a přechody, které byly provedeny v konečném automatu.
- **mnohonásobný FSM coverage** - kombinuje několik FSM dohromady do jednoho modelu pokrytí a měří události, zaměřené na vztahy mezi nimi.

Největší výhodou této metriky je, že nevyžaduje žádny HDL kód navíc, protože nástroje strukturního pokrytí jsou již běžně součástí simulátorů. Tyto nástroje jednoduše vytvoří strukturní pokrytí analýzou HDL kódu DUT. Tato metrika by měla vždy dosahovat po ukončení verifikace 100%, neboť netestovaný kus kódu může obsahovat chyby, nebo se může v lepším případě jednat o tzv. mrtvý kód, tedy kód který je zbytečný, nikdy se nepoužívá a může být tedy bez ovlivnění funkcionality DUT odstraněn.

### Funkční pokrytí

Na rozdíl od strukturního pokrytí neexistuje způsob jak vytvořit funkční pokrytí automaticky. Slouží pro zaměření vytvářených verifikačních testů na určité funkční oblasti implementovaného DUT. Je na pochopení návrháře nebo verifikačního inženýra na co je třeba se v implementovaném hardwarovém systému zaměřit. Jedná se převážně o výskyt určitých hodnot na signálech DUT nebo jeho registrech, případně se může jednat o sledování hodnot několika signálů či registrů v závislosti na sobě. Dalším sledovaným aspektem může být sekvence hodnot, kdy je žádoucí verifikovat přechody mezi určitými stavy.

Pro definice funkčního pokrytí v jazyce SystemVerilog slouží konstrukce *covergroup*, jenž zaobaluje specifikace modelu pokrytí, v podobě bodů pokrytí tzv. *coverpointů*, které implementují funkční pokrytí, a v podobě *cross* konstruktů, které zachytávají souběžné dění na více signálech, tedy ve více bodech pokrytí najednou. Coverpointy jsou strukturami jenž v sobě obsahují jeden či více *binů*, jenž definují hodnoty, skupiny hodnot nebo sekvence hodnot, které mají být během funkční verifikace pokryty. Biny lze definovat ručně nebo je lze nechat generovat automaticky. Příklady několika coverpointů obsahující různé typy definovaných binů, lze vidět na obrázku 2.3 (tato práce není učebním materiálem a nejsou zde tedy vysvětleny všechny možné podoby coverpointů).

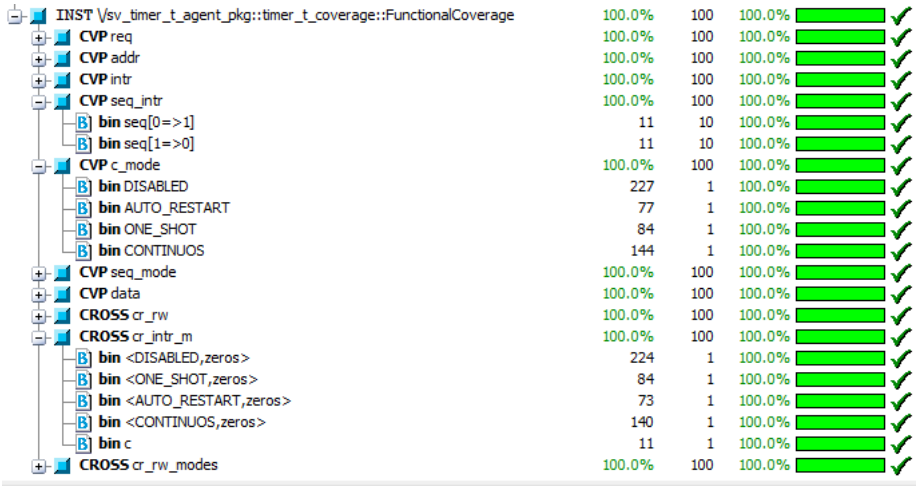
```
bit [3:0] sig_a, sig_b

covergroup cg @(posedge clk);

a: coverpoint sig_a
{
        bins a1 = {0:7};
        bins a2 = {8:15};
}

b: coverpoint sig_b
{
        bins b1 = {0};
        bins b2 = {1:15};
}

c: cross sig_a, sig_b
{
        bins c1 = binsof(a.a2) && binsof(b.b2);
}

trans: coverpoint sig_a
{
        trans1 = (0 => 1), (0 => 2), (0 => 3);
}
```

Obrázek 2.3: Ukázka coverpointů implementovaných v jazyce SystemVerilog

Takto definované body pokrytí je pak možné sledovat v simulačním nástroji. Příklad zobrazení stavu pokrytí je na obrázku 2.4. V prvním sloupci jsou vyobrazeny coverpointy, které jakožto struktury po rozbalení ukazují jednotlivé biny jenž obsahují. Další sloupec pak ukazuje počet zásahů daného bodu pokrytí během simulace. Třetí sloupec zobrazuje počet zásahů, které jsou pro tento bod pokrytí vyžadovány verifikačním inženýrem.

### 2.2.2 Pseudo-náhodné generování stimulů

**Přímé testy** jsou ručně psané a omezené pouze představivostí verifikačního inženýra, který je vytváří. Je samozřejmostí, že tyto verifikační testy se soustředí na položky uvedené ve specifikaci. Jejich největším problémem je časová náročnost rostoucí se složitostí DUT, kdy je potřeba pro dosažení požadovaného pokrytí při verifikaci komplexního DUT vytvořit mnoho přímých testů. Další nevýhodou těchto verifikačních testů jsou nepředvídatelné




Obrázek 2.4: Coverpointy zobrazené v simulátoru ModelSim

chyby, které vznikají nepředvídanou posloupností stimulů na vstupech. Takto psané verifikační testy jsou velice náročné na údržbu, protože jedna změna v DUT si může vyžádat ruční úpravu ve všech implementovaných přímých testech.

S narůstající komplexností hardwarových systémů nám v oblasti verifikace přestávají stačit pouze přímé testy. Využití zcela náhodných testů se jeví jako dobré řešení, ale je zde problém s jednou z verifikačních výzev a to konkrétně stavový prostor verifikovaného systému. Pro zcela vyčerpávající verifikaci hardwarového systému by musel verifikační inženýr projít všechny možné stavy, možné kombinace vstupů a rovněž by také musel brát do úvahy i všechny přechody mezi aktuálním a následujícím stavem. Naneštěstí i jednoduchý hardwarový obvod může mít obrovský stavový prostor [26]. Pokud bychom chtěli pouze pomocí náhodného generování stimulů ověřit všechny tyto stavy, trvalo by to velice dlouho nehledě na generování redundantních stimulů. Funkční verifikace proto zavedla tzv. pseudonáhodné generování stimulů (angl. *pseudo-random stimulus generation*), na které jsou uplatněna různá omezení jako například podmínka pro generování pouze validních vstupů nebo například při stejném chování DUT na určitou množinu vstupních hodnot generovat jen jednu hodnotu z této množiny atd. To dalo vzniknout tzv. pseudonáhodným testům.

**Pseudonáhodné testy** jsou rychlejší na implementaci, není třeba vymýšlet posloupnosti stimulů pro testování, generátor na základě omezení generuje velké množství stimulů načež je takto vygenerováno i mnoho přímých testů, které by jinak musel verifikační inženýr psát ručně. Výhodou těchto verifikačních testů je rychlé pokrývání stavového prostoru. Hlavní nevýhodou těchto verifikačních testů je generování redundantních stimulů a jejich posloupností a může proto trvat delší dobu než je dosaženo 100% pokrytí za využití pouze pseudonáhodných testů. Toto je jeden z problémů jímž se zabývá tato práce.

Posledním typem verifikačního testu zde uvedeným je tzv. **negativní test**. Jak bylo uvedeno výše, je generování stimulů různě omezováno podle potřeby. Ve skutečném světě nelze zaručit pouze správně formátovaná a zcela předvídatelná data na vstupech. Kromě normální funkcionality DUT je správné zvládání chyb známkou robustnosti, což je další klíčový faktor pro zlepšení kvality výsledného výrobku. Z tohoto důvodu je důležité vyzkoušet i chybové vstupy, aby byla ověřena reakce DUT na ně. Nelze si při vytváření omezujících podmínek pro pseudonáhodné generování stimulů dovolit nechat generovat jen správné vstupy a ignorovat vstupy chybové.

### 2.2.3 Verifikace založená na formálních tvrzeních

Úkolem této přídavné techniky je formalizovat předpoklady o chování DUT, které musí být pravdivé po celou dobu běhu systému. Jedná se o jednoduchá formální tvrzení, která lze klasifikovat jako statická kontrolující absenci nechtěné události, bez závislosti na jakýchkoliv jiných událostech, které mohou nastat. Lze také tvořit více-cyklové nebo komplexní tvrzení například pro kontrolu dodržení určitého protokolu uvnitř DUT nebo pro kontrolu komunikace mezi jednotlivými bloky DUT a mnoho dalších. Stále větší využívání formálních tvrzení v hardwarové verifikaci si vyžádalo vytvoření speciálních jazyků pro jejich zápis jako například PSL (angl. *Property Specification Language*) nebo SVA (angl. *SystemVerilog Assertions*) [26].

### 2.2.4 Samokontrolní mechanismy

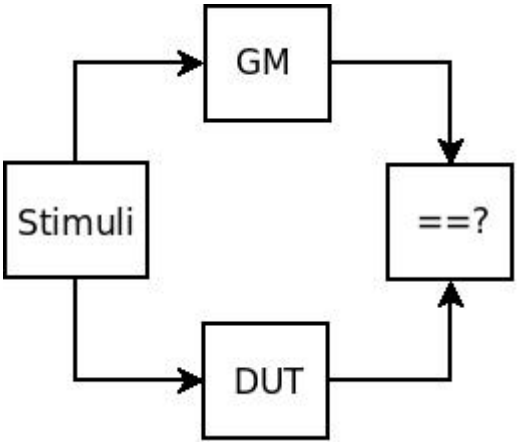
Tato technika slouží k zautomatizování kontroly výstupů DUT. Pro automatickou kontrolu výstupů lze využít jeden ze dvou základních přístupů, kterými jsou referenční vektory a referenční model.

**Referenční vektory** jsou seznamem očekávaných hodnot na výstupu, které jsou před během verifikace nahrány do verifikačního prostředí. Během verifikace dochází k porovnávání výstupů DUT s těmito vektory. Tato kontrola se může provádět v každém cyklu nebo pro každou transakci. Verifikační inženýr vytváří referenční vektory ručně nebo je generuje pomocí nějakého externího programu a nebo je možnost jejich obdržení od zákazníka. Jednou z největších nevýhod je udržovatelnost referenčních vektorů, protože při jakékoliv změně chování DUT je třeba provést úpravy na generování referenčních vektorů nebo v referenčních vektorech samotných při ruční tvorbě, což může verifikaci značně časově prodloužit pokud dochází k častým změnám či opravám DUT. Dalším problémem je samotná jejich tvorba, ruční tvorba referenčních vektorů pro složitější systémy může být zdlouhavá a i když je využit externí program pro jejich generování, je nejdříve třeba tomuto programu předat znalost o tom jak DUT pracuje [26].

**Referenční model** nebo **Golden model** (dále bude využíváno zkratky GM) reimplementuje funkci DUT na základě jeho specifikace v nějakém vyšším programovacím jazyce nebo v HVL. Jedna z možných implementací GM provádí dynamickou transformaci vstupů na výstupy v každém hodinovém cyklu, známá také pod označením *"cycle-accurate"* model [26]. Tento způsob automatické kontroly pracuje tak, že stejné vstupy jsou přivedeny jak na vstupy GM tak na vstupy DUT a následně dochází k jejich porovnání 2.5. Udržovatelnost GM při změně chování DUT není tak časově náročná jako u referenčních vektorů. Nevýhodou tohoto přístupu je náročnost implementace GM, která může trvat relativně dlouho pro komplexnější hardwarové systémy a také je třeba zajistit, aby GM dodržoval interní časování DUT.

## 2.3 SystemVerilog

SystemVerilog je jazyk pro popis hardwarových obvodů, ale zejména je určen pro jejich verifikaci skrze své rozšíření. Tímto se zařadil SystemVerilog mezi HDL i HVL jazyky. Historie tohoto jazyka sahá do devadesátých let minulého století, kdy HDL jazyk Verilog se stal nejvíce rozšířeným jazykem pro popis hardwaru, simulaci a syntézu. Brzy se však stal nedostačujícím pro verifikaci velkých hardwarových obvodů a proto byly vytvořeny




Obrázek 2.5: Kontrola výstupů DUT pomocí GM

komerční HVL jazyky jako *OpenVera* a *e*. Společnosti, které nechtěly platit za tyto komerční nástroje trávily roky vytvářením vlastních nástrojů.

Nutnost využívat dvou různých jazyků pro popis hardwaru a jeho následnou verifikaci vytvořila úzké místo v komunikaci mezi týmy návrhářů a verifikačních inženýrů. To vše vedlo ke vzniku organizace Accellera, která chtěla vytvořit novou generaci jazyka Verilog [23].

V letech 1997 až 2002 byl vyvíjen jazyk *Superlog*, který měl vyřešit problém jednotného jazyka pro návrh i verifikaci hardwarových obvodů. V roce 2001 byla HDL část Superlogu vystavená na jazyce Verilog, předána organizaci Accellera, aby se stala standardem. Tento standard byl však pojmenován **SystemVerilog** jako reakce na vytvoření standardu s názvem SystemC. V roce 2002 byly začleněny do tohoto standardu některé HVL funkce z jazyka *Vera* a následně v roce 2005 také některé HVL funkce z jazyka *OpenVera*. V listopadu 2005 organizace Accellera dosáhla svého cíle a vznikl standard IEEE 1800-2005 pro SystemVerilog [18].

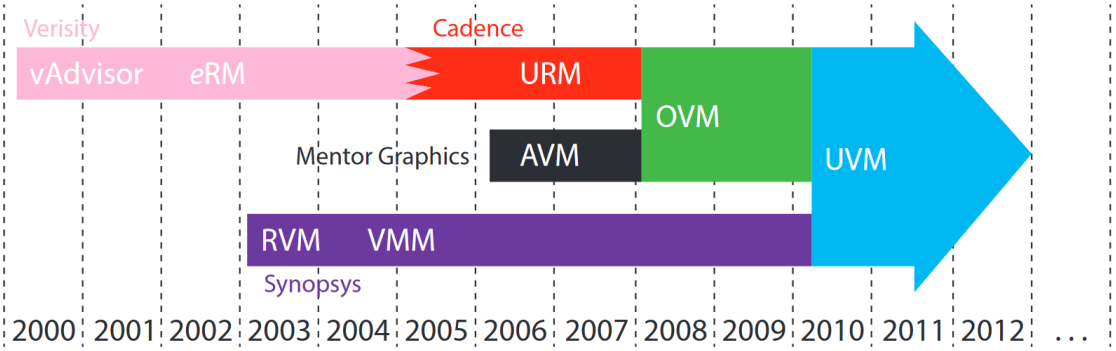
Největším přínosem jazyka SystemVerilog pro verifikaci se stala možnost využít prvky objektově-orientovaného programování (angl. *Object-oriented programming*, OOP). Stejně jako u jakýchkoliv jiných OOP programovacích jazyků jsou definice tříd šablony pro objekty vytvářené v paměti. Jakmile jsou vytvořeny, objekty zůstavají v paměti dokud nejsou odstraněny všechny odkazy na ně a následně odstraněny z paměti pomocí procesu *"garbage collector"* běžícím na pozadí. Třídní šablony definují členy, kterými mohou být datové proměnné nebo metody. V jazyce SystemVerilog jsou dva druhy metod, jedním z nich je funkce, která nespotřebovává čas (bere se tedy, že je celá funkce provedena za nulový čas) a druhým typem metod jsou úkoly (angl. *tasks*), které spotřebovávají simulační čas. Protože třída je objekt, který musí být vytvořen předtím než existuje v paměti, musí být vytvoření třídní hierarchie ve verifikačním prostředí psaném v SystemVerilogu inicializováno v modulu, který je statickým objektem a je přítomen na začátku simulace. Ze stejného důvodu třída nemůže obsahovat modul. Třídy jsou považovány za dynamické objekty, protože se mohou vytvářet a rušit za běhu simulace [21].

Zavedení OOP umožnilo využití klasických technik tohoto typu programování jako jsou třídy, zapouzdření, dědičnost a polymorfismus na poli verifikace. Díky tomu se zvýšila modularita verifikačního prostředí, což usnadňuje hledání chyb. Je možné znovupoužití již dříve implementovaných komponent verifikačního prostředí a tím usnadnění tvorby nových. Dovoluje to také vytvářet verifikační prostředí na vyšší úrovni abstrakce, není třeba nadále

měnit jednotlivé bity ručně, místo toho je možné zavolat rutinu, ve které je tato činnost implementována. Všechny tyto vlastnosti mají obrovský vliv na efektivitu verifikace. Pokud vlastnosti jazyka SystemVerilog jsou pro danou úlohu nedostačující, je zde možnost využít jiný jazyk skrze *"Direct programming interface"* (DPI). Toto rozhraní umožňuje SystemVerilogu volat funkce implementované v jiných jazycích. Kromě rozšiřujícího aspektu z pohledu funkcionality je zde rovněž možnost využít již dříve napsaného kódu pro verifikaci DUT v jiném programovacím jazyce. Jako poslední je zde uvedeno rozšíření jazyka SystemVerilog o jazyk pro psaní formálních tvrzení SVA. Díky tomu, že je součástí jazyka SystemVerilog je snadné zakomponovat formální tvrzení do verifikačního prostředí psaného v tomto jazyce.

## 2.4 Verifikační metodiky

Verifikační metodiky přinášejí další výrazná vylepšení do procesu funkční verifikace. Jejich hlavním cílem je vytvoření modulární, znovupoužitelné architektury a stimulů pro verifikační prostředí skrze vytváření standardizovaných třídních knihoven. Tyto třídy pak slouží jako základ pro vývoj a tvorbu verifikačních prostředí pro funkční verifikaci. Na obrázku 2.6 je ukázán vývoj těchto metodik v čase.




Obrázek 2.6: Vývoj metodik v čase [3]

V této kapitole je rozebrána pouze metodika UVM, která je v této práci využita. Mezi další známé metodiky, které je možné zmínit, patří Open Verification Methodology (OVM) a Verification Methodology Manual (VMM).

### 2.4.1 UVM

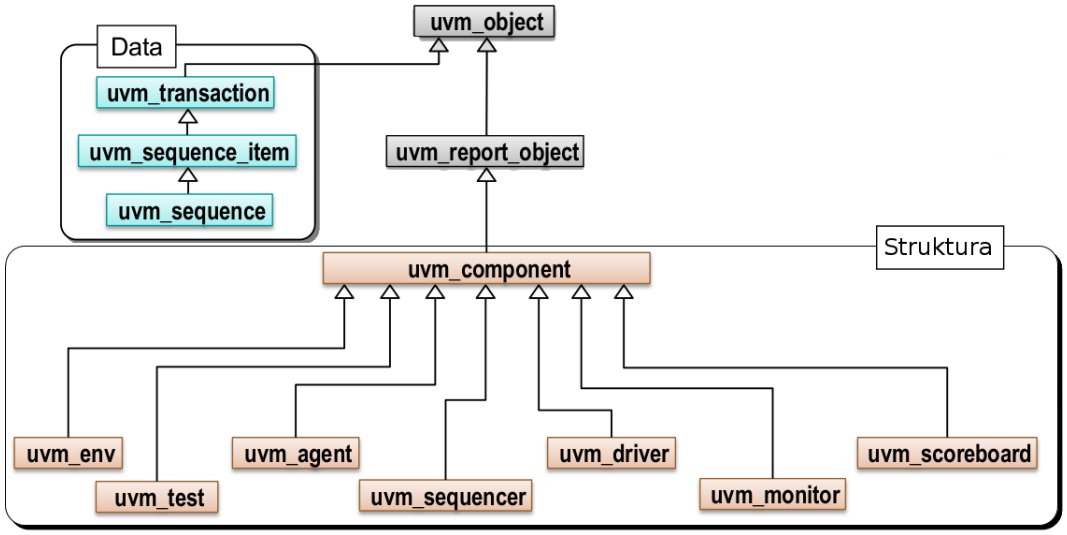
UVM je zkratka pro Universal Verification Methodology a je standardem organizace Accellera. Tato metodika specifikuje, jak vytvářet znovupoužitelné a snadno rozšiřitelné verifikační prostředí pro hardwarové návrhy psané v jazycích SystemVerilog, Verilog, VHDL a SystemC. UVM samotné je knihovna základních tříd (angl. *Base Class Library*, BCL), které usnadňují tvorbu strukturovaných verifikačních prostředí s využitím volně dostupného kódu (angl. *open source*), které mohou běžet na jakémkoliv SystemVerilog IEEE 1800 simulátoru. Jedná se o první standard, který spolu s dokumentací šíří také zdrojové kódy jako volně dostupné a to pod licencí Apache [14] [21].

Hned na začátku je třeba zdůraznit některé aspekty, na kterých si metodika zakládá. Verifikační prostředí vytvořená pomocí této metodiky využívají pseudonáhodné generování stimulů a samotná verifikace je řízena funkčním pokrytím. Dalším důležitým aspektem je

konfigurovatelnost a flexibilita vytvořených verifikačních prostředí a v neposlední řadě je velice důležitá znovupoužitelnost verifikačních komponent, z nichž je verifikační prostředí sestaveno. Všechny tyto zmíněné aspekty dávají představu o tom jaké další výhody metodika UVM poskytuje [14]:

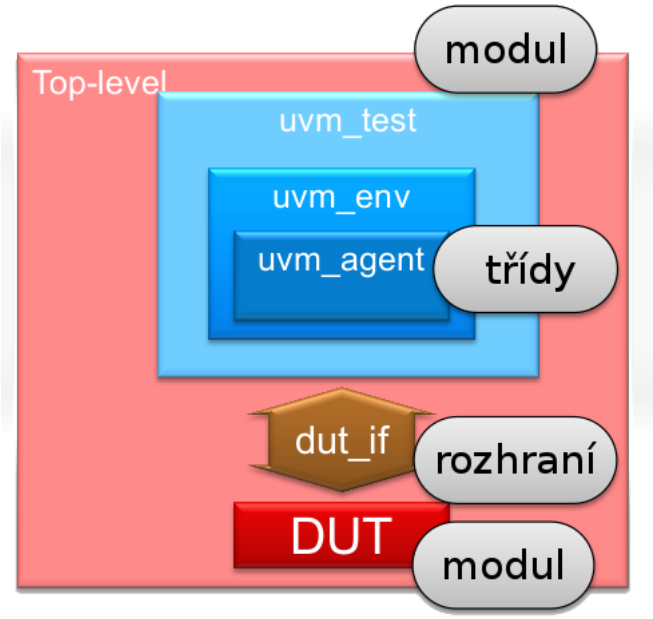
- **Oddělení verifikačních testů** (verifikačních scénářů) od verifikačního prostředí.
- **Komunikaci na úrovni transakcí** (transakce jsou vysvětleny dále v textu), což zvyšuje míru abstrakce na které verifikační komponenty komunikují mezi sebou. Vede to také k odstranění některých detailů, které ztěžovaly znovupoužitelnost verifikačních komponent.
- **Vrstvené sekvenční stimuly**. Toto slovní spojení znamená, že kromě možnosti pseudonáhodně generovat stimuly, je možno díky vrstvenému modelu mít sekvence sekvencí stimulů, což přináší možnost koordinovat přiváděné stimuly na úrovni rozhraní.
- **Standardizované** poskytování reportů během simulace.

Postupně jsou na následujících stránkách popsány základní třídy metodiky UVM a jejich hierarchická struktura, některé komponenty verifikačního prostředí a jsou zmíněny také základní fáze, jakými komponenty verifikačního prostředí procházejí.




Obrázek 2.7: Hierarchie tříd metodiky UVM [13]

Z obrázku 2.7 je možno vyčíst, že UVM knihovnu lze rozdělit do několika typů tříd, kde každý typ třídy se stará o jiné aspekty verifikačního prostředí. Prvním z typů jsou **objekty**, jejichž základní třídou je rovněž také kořenová třída knihovny `uvm_object`, ze které dědí všechny ostatní třídy. Objekty se používají jako datové struktury pro konfiguraci verifikačního prostředí. Základní třídou dalšího typu je `uvm_transaction` a nese označení **data**. Tento typ se využívá pro generování a analýzu stimulů. Třídy v tomto typu se liší od ostatních tak, že jejich instancí je mnoho vytvářeno a rušeno napříč celou simulací. Posledním typem je pak **struktura**, jejíž základní třídou je `uvm_component`. Tento typ má na starosti konstrukci hierarchické struktury verifikačního prostředí založeném na třídách [11].




Obrázek 2.8: Hierarchická struktura verifikačního prostředí podle UVM [12]

Hierarchická struktura verifikačního prostředí je ukázána na obrázku 2.8. Celá verifikace založená na metodice UVM (verifikační prostředí, DUT a rozhraní, které zajišťuje jejich komunikaci) je uzavřená do jediného modulu, který po zbytek kapitoly budeme jednoduše označovat jako **top modul** (angl. *top-level module*). Tento modul také provádí spouštění samotného testování ve svém inicializačním bloku voláním UVM metody `run_test` s názvem testu jako parametrem (ve formě řetězcového literálu), který se má provést. Samotné verifikační prostředí je složeno ze dvou částí, jimiž jsou **Test** a **Prostředí** (angl. *environment*, zkratka Env odvozená z anglického výrazu je využívána na obrázcích). Část **Test** odvozená od třídy `uvm_test` tvoří proměnou část verifikačního prostředí a část **Prostředí** odvozená od třídy `uvm_env` tvoří jeho pevnou část [16].

Na nejvyšším stupni hierarchické struktury verifikačního prostředí je komponenta **Test**, která má na starosti konfiguraci verifikačního prostředí, inicializaci konstrukčního procesu vybudováním další úrovně komponent a inicializaci stimulů spuštěním hlavní sekvence. Specifikuje například, jaké sekvence se mají provést, nebo podle jakého pokrytí má být simulace řízena. Těchto komponent může být více, může být vytvořena celá jejich knihovna a pak je možné jednoduše vybrat jednu z nich podle potřeby při volání UVM metody `run_test` [21].

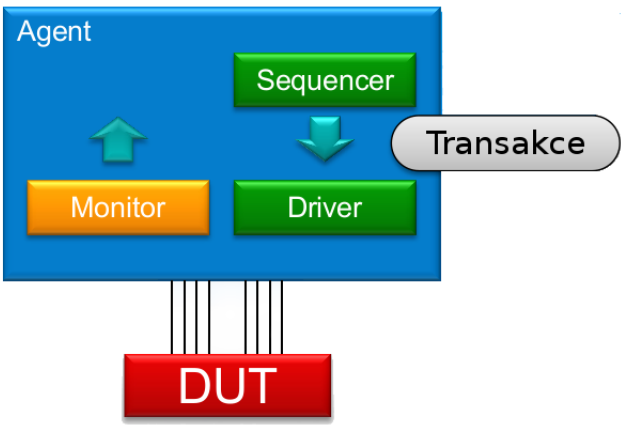
Hierarchii strukturní části verifikačního prostředí si lze představit jako sérii obálek, kde je přesně definováno, které komponenty budou umístěny ve kterých. Architektura UVM verifikačního prostředí je modulární pro usnadnění znovupoužití skupin verifikačních komponent v dalších projektech nebo na vyšším stupni integrace stejného projektu. Hlavní dva typy obálek, které usnadňují znovupoužití jsou **Prostředí** a **Agent** odvozený od třídy `uvm_agent`.

Než je přikročeno k dalšímu výkladu je zde vysvětlen typ tříd uvedený na obrázku 2.7 pod názvem **data**. Na vrcholu tohoto typu je třída `uvm_transaction` a hned je také možné zmínit třídu `uvm_sequence_item`. Obě tyto třídy slouží jako základní pro tzv. **transakce**. Transakce tvoří obálku pro vstupní data DUT. Obsahuje množinu dat nebo atributů

v závislosti na modelovaném protokolu DUT. Může také obsahovat množinu omezujících podmínek pro zajištění generování podstatných či důležitých hodnot. Omezující podmínky uvnitř transakce jsou brány jako základní a mohou být vypínány, když je například potřeba aplikovat jiná omezení na transakci definovaná mimo ni. Pro tvorbu transakcí se využívá jen `uvm_sequence_item`, protože obsahuje navíc infrastrukturu pro umožnění komunikace na úrovni transakcí mezi sequencerem a driverem. Poslední třídou typu data je `uvm_sequence`, od které jsou odvozovány **sekvence**. Sekvence se stará o postupné generování jednotlivých transakcí. Jakým způsobem a kolik transakcí bude generováno je čistě v rukou verifikačního inženýra. Lze vytvářet ručně psané transakce, je možné využít náhodného generování a je také možné ovlivňovat transakce mezi sebou. Tato třída je parametrizovaná typem transakce, které bude generovat [13].

**Agent** je v UVM metodice označení pro verifikační komponentu. Většina DUT má více různých signálových rozhraní, každé s jiným komunikačním protokolem. Agent slouží k tomu, aby poskytl možnost zasílat na tato rozhraní data a rovněž poskytl možnost zasílaná a přijímaná data monitorovat. Pro každé rozhraní na DUT je vytvořen jeden agent z čehož plyne, že agenti jsou specifičtí pro protokol, který je na daném rozhraní [21]. Agent je složen obvykle ze tří dalších strukturních komponent (viz 2.9), jimiž jsou:

- **Driver** - strukturní komponenta driver je odvozena od třídy `uvm_driver` a je zodpovědná za převod dat v sekvencích transakcí na signálovou úroveň pinů DUT a zpět.
- **Monitor** - strukturní komponenta monitor je odvozena od třídy `uvm_monitor` a provádí monitorování signálů na pinech DUT, které následně převádí do podoby transakcí, jenž přeposílá do analytických komponent.
- **Sequencer** - strukturní komponenta sequencer je odvozena od třídy `uvm_sequencer` a má na starosti přesměrovávání transakcí generovaných v sekvencích z/do komponenty driver (je zde možnost zpětného ovlivňování generace dalších transakcí, proto ta možnost přeposílání z driveru).




Obrázek 2.9: Komponenta Agent metodiky UVM [13]

Činnost agenta lze jednoduše popsat následujícím způsobem. UVM objekt sekvence postupně generuje, vždy na vyžádání strukturní komponenty driver, novou transakci. Transakce je pomocí sequenceru přeposlána do driveru, který v ní obsažená data převede na signálovou úroveň pinů DUT a pošle je přes rozhraní do DUT. Agent rovněž pomocí své

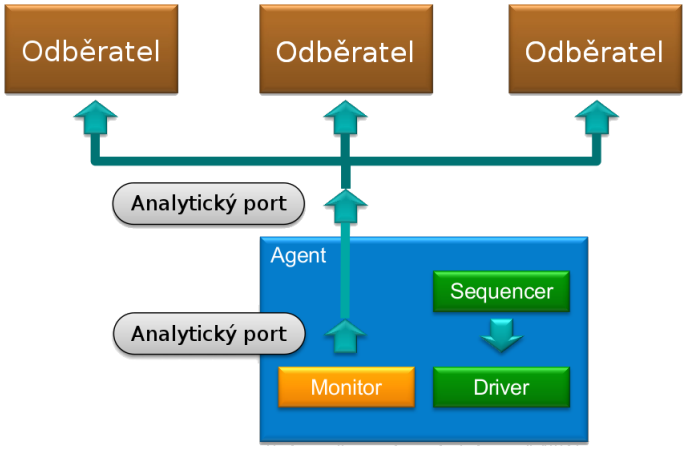
komponenty monitor zachytává signály na vstupech a výstupech DUT, které následně převede na transakční úroveň a přeposílá je do analytických komponent.

Propojení verifikačního prostředí s DUT se neděje přímo skrze rozhraní, ale je vytvořeno rozhraní virtuální, které je ukazatelem na instanci rozhraní připojenou k DUT. Virtuální rozhraní je použito, protože verifikační prostředí založené na třídách se nemůže odkazovat na Verilog/VHDL porty a nemůže se tedy přímo připojit k DUT. Verifikační prostředí může monitorovat a kontrolovat piny DUT nepřímo skrze prvky virtuálního rozhraní [21]. Virtuální rozhraní je vytvořeno v top modulu a je postupně skrze hierarchii využíváno v různých komponentách, zejména v driveru a monitoru [12].

**Prostředí** slouží k seskupení všech agentů komunikujících s DUT na jedno místo. Kromě agentů může obsahovat ještě některé nebo všechny z následujících typů komponent:

- **Scoreboards** - strukturní komponenty scoreboards jsou odvozeny od třídy `uvm_scoreboard` a patří mezi analytické komponenty. Scoreboard kontroluje zda se DUT chová správně a to tak, že porovnává transakce z DUT a referenční transakce.
- **Predictors** - tento typ komponenty byl v této práci uveden také jako **golden model** nebo **referenční model**. Slouží pro dynamickou transformaci stejných vstupních transakcí jako obdržel DUT a následně dochází k porovnávání jejich reakcí. Pro rozsáhlejší popis viz 2.2.4.
- **Monitory funkčního pokrytí** - tento typ komponent patří mezi analytické komponenty. Komponenty tohoto typu slouží pro odpovídání na otázku *"Už je možné ukončit verifikaci?"*. Tyto komponenty sbírají data a ukládají je do sdílené databáze pokrytí, která ukazuje aktuální stav verifikace DUT. Tento typ komponent je obvykle specifický pro dané DUT.

Mimo výše uvedených komponent může prostředí obsahovat i další analytické komponenty. Tyto komponenty (na obrázku 2.10 jsou vyobrazeny jako **odběratelé**) se připojují k analytickému portu monitoru, který jim přes něj hromadně rozesílá získané transakce. K analytickému portu může být připojen libovolný počet analytických komponent i žádná.

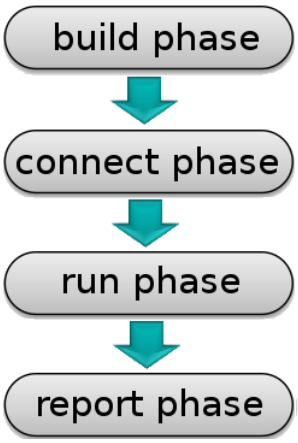



Obrázek 2.10: Připojení komponent přes analytický port k monitoru [15]

**Fáze** slouží v metodice UVM pro řízení běhu verifikačních scénářů ve verifikačním prostředí. Základní třída pro strukturu verifikačního prostředí `uvm_component` obsahuje

virtuální metody pro jednotlivé fáze, které jsou postupně volány podle potřeby pro každou strukturní komponentu, kde jsou implementovány. Zavedení fází umožňuje vytvářet jednotlivé verifikační komponenty samostatně, protože každá z těchto komponent danou fázi implementuje podle zavedené definice [11]. Fáze jsou rozděleny do několika skupin, ze kterých je vybráno a dále popsáno několik základních (tyto fáze je možné také vidět na obrázku 2.11):

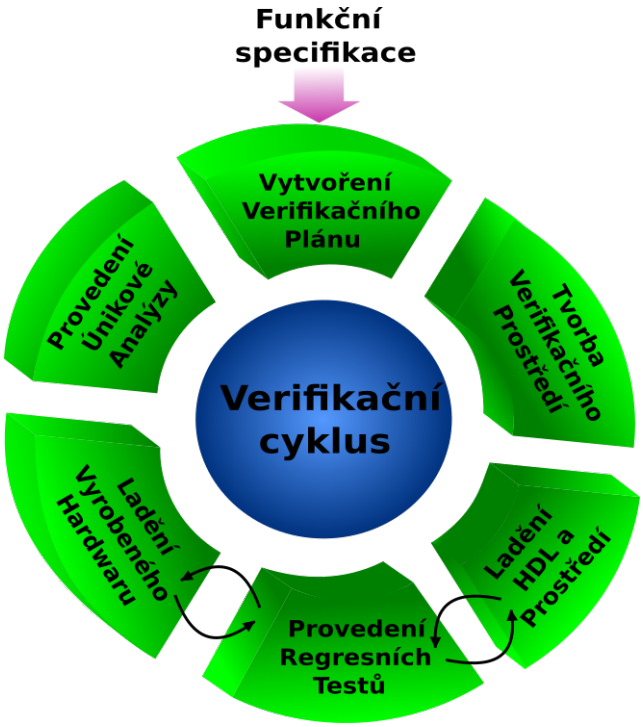
- **Fáze build** (angl. *build phase*) vytváří instance jednotlivých verifikačních komponent. Tato fáze je také označovaná jako shora dolů, protože nejdříve jsou vytvořeny v metodě build fáze instance komponent, které daná komponenta obsahuje (pokud nějaké obsahuje) a následně jsou volány metody build fáze pro tyto jednotlivé komponenty. Každá komponenta je vytvářena tak, aby bylo možné ji konfigurovat komponentou v hierarchii o úroveň výš. Instance komponent jsou vytvářeny nepřímo pomocí továrních metod UVM.
- **Fáze connect** (angl.*connect phase*) provádí propojování jednotlivých komponent verifikačního prostředí pro komunikaci na úrovni transakcí jako například připojení analytických komponent na analytický port ukázané na obrázku 2.10.
- **Fáze run** (angl. *run phase*) metoda této fáze je jako jediná implementována formou tasku (viz 2.3), tedy jako jediná spotřebovává simulační čas. Tato fáze je vykonávána paralelně na všech komponentách verifikačního prostředí, které ji obsahují. Je používána pro generování stimulů a sledování chování DUT, jedná se o fázi kde probíhá samotná simulace.
- **Fáze report** (angl. *report phase*) je provedena po dokončení simulace k zobrazení dosažených výsledků. Tuto fázi většinou používají analytické komponenty.




Obrázek 2.11: UVM fáze

## 2.5 Proces verifikace

Nejdůležitější otázkou pro verifikačního inženýra zůstává, kdy může verifikaci ukončit, tedy jestli je už dostatečně ověřeno DUT a jeho chování odpovídá zadané specifikaci. Na základě [26] jsou uvedeny v této kapitole hlavní kroky verifikace, které je také možné vidět na obrázku 2.12.




Obrázek 2.12: Verifikační cyklus [26]

### 2.5.1 Specifikace

Základem celého verifikačního procesu je specifikace popisující požadavky na finální produkt. Obsahuje například specifikaci rozhraní, pomocí kterých má komunikovat se svým okolím, popis funkce, kterou má provádět a další aspekty, které ovlivňují finální produkt. Jak verifikační tým tak tým návrhářů by měl provádět realizaci produktu podle specifikace zcela odděleně. Může se to zdát jako zbytečné, ale druhá implementace dopomáhá oběma týmům ověřit zda byla zadaná specifikace pochopena správně.

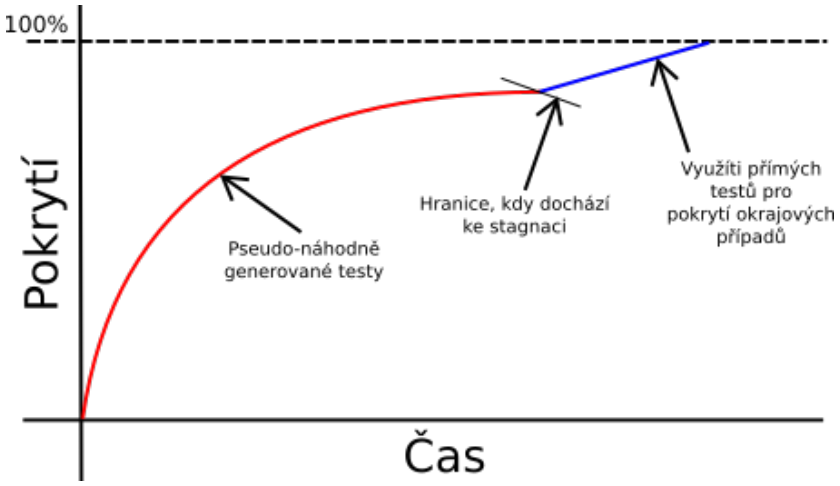
### 2.5.2 Verifikační plán

Verifikační plán je pro celý proces zásadní. Dopomáhá verifikačním inženýrům si uvědomit co se bude verifikovat a jakým způsobem se to bude verifikovat. Pokud se nejedná o jednoduchý návrh je vhodné využít hierarchickou strukturu návrhu i při jeho verifikaci. Začít s verifikací menších komponent a postupně se propracovat až k verifikaci na systémové úrovni. Postupně jsou zde uvedeny některé prvky, které má verifikační plán obsahovat.

- Jednou z věcí, které by měl verifikační plán určitě obsahovat, je určení toho jakým způsobem a jaké stimuly budou generovány. Rozlišujeme tři typy verifikačních testů:
  - **Přímé testy**
  - **Pseudo-náhodné testy**
  - **Negativní testy**

  Ve většině případů se kombinují přímé a pseudonáhodné testy pro dosažení 100% pokrytí v co nejkratším čase. Začíná se pseudonáhodnými testy, kdy následně při

stagnaci těchto verifikačních testů na určitém procentu pokrytí, je vytvořen jeden či více přímých testů pro dosažení 100% pokrytí. Přímé testy jsou většinou potřeba pro okrajové případy, kterých se dosahuje za pomoci pseudonáhodných testů obtížně viz graf 2.13.




Obrázek 2.13: Pokrytí v závislosti na čase

- Součástí verifikačního plánu je také určení jakým způsobem budou prováděny kontroly výstupních dat DUT. Pro kontrolu se využívají samo-kontrolní mechanismy viz 2.2.4 a také formální tvrzení viz 2.2.3
- Dále je třeba určit, kdy budou výsledky provedených verifikačních scénářů kontrolovány. Existují dva hlavní přístupy kdy provádět kontrolu výsledků.
  - **Kontrola výstupů za běhu** (angl. *on-the-fly-checking*) je nejlépe použitelná pro DUT, které provádí nějaký typ transformace na vstupních transakcích. Tento přístup je jednodušší na implementaci a také hledání chyb je s ním jednodušší, protože simulace zastaví jakmile je detekována chyba. Simulace využívající tohoto přístupu také potřebuje daleko méně paměti, neboť není třeba si pamatovat veškeré předpokládané výstupy za dobu celé simulace (scoreboard může transakci okamžitě smazat po provedení porovnání s referenční transakcí).
  - **Kontrola výstupů na konci verifikačního testu** (angl. *end-of-test-checking*) je většinou využita pokud nastane jedna z následujících podmínek:
    * Výsledky zůstávají trvale v paměti DUT až do konce verifikačního testu.
    * Je omezen přistup k signálům například z důvodů akcelerace nebo využití emulace.
    * Je potřeba provést kontrolu koncového stavu DUT a/nebo verifikačního prostředí.
    * Funkce DUT mají systémové aspekty.

    Kontrola při tomto přístupu může být provedena i pomocí externího programu pro analýzu výsledků. Stačí aby získané výsledky byly zapsány do souboru místo do paměti.
- Jako poslední důležitý bod verifikačního plánu jsou zde zmíněna **kritéria pro ukončení verifikace**, která definují metriky určující úplnost verifikace.

### 2.5.3 Konstrukce verifikačního prostředí

Existuje mnoho typů verifikačních prostředí. Každé z těchto prostředí používá jiný způsob pro generování stimulů a kontrolu výstupů DUT. Ve všech případech je nutné mít s čím porovnávat výstupy DUT pro jejich kontrolu. K tomuto účelu slouží samo-kontrolní mechanismy viz 2.2.4. V případě GM verifikační inženýři promítají pochopení zadané specifikace do jeho tvorby. Prostředí je neustále zdokonalováno po celou dobu verifikačního procesu.

### 2.5.4 Ladění HDL a verifikačního prostředí

V tomto kroku procesu verifikace dochází k integraci verifikačního prostředí s DUT. Provádějí se připravené verifikační testy, při kterých verifikační inženýři nacházejí různé anomálie, které po přezkoumání odhalí chybu buď ve verifikačním prostředí nebo DUT. Anomálie nastává když verifikační prostředí předpoví jiný výstup než DUT. Po odstranění chyby je verifikační test opakován pro kontrolu, zda byla chyba úspěšně opravena a také jestli nebyla vytvořena při opravě chyba nová. Tento postup se opakuje dokud neprojdou všechny verifikační testy bez chyb.

### 2.5.5 Regrese

Regrese je nepřetržitý běh verifikačních scénářů definovaných ve verifikačním plánu. Tento krok je v procesu verifikace ze dvou důvodů. Prvním je náhodnost generování stimulů zabudovaná do verifikačního prostředí, která umožňuje generování mnoha rozdílných verifikačních scénářů při každém spuštění verifikačního testu (stačí změnit tzv. *seed* pro generování). Druhým důvodem opakování všech definovaných verifikačních testů jsou opravy provedené na DUT. Pro odhalení špatně odhalitelných chyb se zvyšuje počet pracovních stanic, na kterých probíhá verifikace. Na každé pracovní stanici mohou díky náhodnosti být prováděny jiné verifikační scénáře. Při odhalení chyby se postupuje stejně jako v předchozím kroku procesu verifikace.

### 2.5.6 Ladění vyrobeného hardwaru (systémový test)

V tomto kroku je provedeno oživení hardwaru, během kterého může dojít k dalším anomáliím. Úkolem verifikace je pokud možno se vyhnout nalezení chyby na již vyrobeném skutečném hardwaru, protože jeho výroba je drahá. Ladění na skutečném hardwaru je daleko obtížnější, neboť nejsou k dispozici výhody trasování jako tomu bylo při testování s využitím verifikačního prostředí.

### 2.5.7 Úniková analýza

Pokud během oživení hardwaru byla odhalena chyba, pak musí verifikační tým provést tzv. **únikovou analýzu** (angl. *escape analysis)*. Cílem tohoto kroku je zajištění, že verifikační tým plně chápe nalezenou chybu a zná důvod proč nebyla tato chyba odhalena během testování ve verifikačním prostředí.

Verifikační tým musí, pokud je to možné, reprodukovat nalezenou chybu ve verifikačním prostředí. Tým nemůže tvrdit, že oprava chyby byla úspěšná, dokud nedokáže reprodukovat skutečnou chybu během verifikace.

# Kapitola 3

# Mentor Questa InFact

V této kapitole jsou blíže vysvětleny portovatelné stimuly a jejich souvislost s nástrojem Questa InFact od společnosti Mentor. Dále jsou popsány některé základní konstrukce a syntaxe jazyka používaného tímto nástrojem a následně je nastíněn způsob jakým tento nástroj umožňuje portovatelnost verifikačních scénářů v něm definovaných.

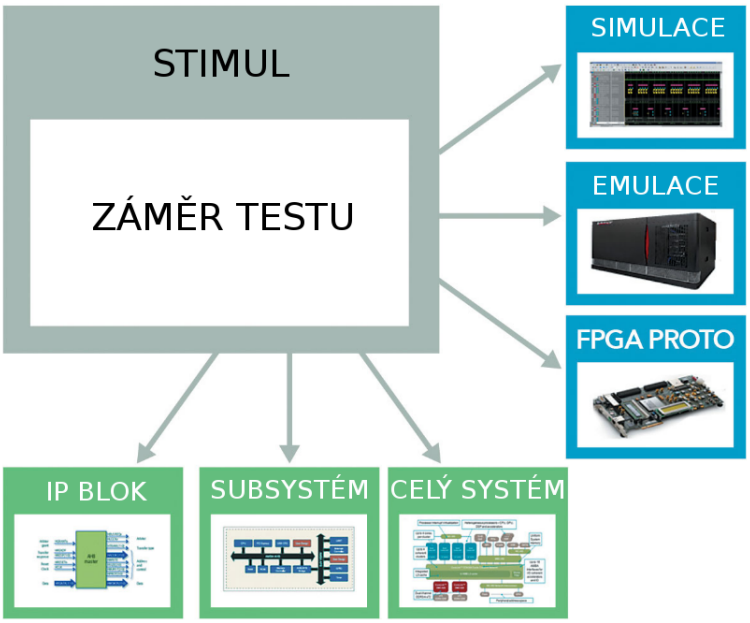
## 3.1 Portovatelné stimuly

Než je přikročeno k popisu samotného nástroje, je uvedeno proč jsou portovatelné stimuly potřeba, co to vlastně jsou portovatelné stimuly a jak nástroj Questa InFact souvisí s novým standardem pro portovatelné stimuly.

V posledních několika letech bylo vynaloženo mnoho úsilí k zvýšení efektivity a produktivity verifikace hardwarových návrhů s důrazem na zachování kvality. Výsledkem je dnes běžně používaná metodika UVM využívající pseudonáhodné a/nebo přímé verifikační testy pro verifikaci hardwarových návrhů (viz 2.5.2). Snahou je využívat pouze pseudonáhodné verifikační testy, které se však neobejdou bez generování značného množství redundantních stimulů pro dosažení všech okrajových stavů DUT. Redundance narůstá exponenciálně s komplexností navrhovaného hardwaru. Zatímco na úrovni IP bloků a subsystémů jsou pseudonáhodné verifikační testy navzdory redundanci úspěšně používány, tak tento přístup ke generování stimulů selhává na systémové úrovni. Na systémové úrovni je potřeba verifikovat mnoho různých prvků a je nutné při verifikaci udržovat jejich správnou koordinaci. Další slabinou je zvyšující se počet hardwarových návrhů zahrnujících jeden či více procesorů, což zavádí potřebu provádět společnou verifikaci softwaru obsaženého v procesorech a hardwaru na systémové úrovni [22] [5].

Portovatelné stimuly vznikly jako další krok ke zvýšení efektivity a produktivity verifikace (více o možnostech zavedených portovatelnými stimuly je v podkapitole 3.2.6). Nejlépe vyjadřuje jejich hlavní myšlenku obrázek 3.1. Slovně je lze popsat jako způsob vyjádření záměrů testování určený k vytvoření verifikačních scénářů s tím, že takto vytvořené verifikační scénáře bude možné opětovně použít při verifikaci na jiných platformách, ve verifikačních prostředích napsaných v různých HVL jazycích a na vyšších úrovních abstrakce.

Portovatelné stimuly zvyšují úroveň abstrakce z transakční úrovně (TLM, angl. *Transaction Level Modeling*) na úroveň scénářů. Nejsou již definovány přímo sekvence transakcí, ale je definováno za pomoci jazyka určeného pro portovatelné stimuly přímo požadované chování v podobě pravidel. Takto definované chování je převedeno do grafové reprezentace, která je dále využívána ke generování stimulů podle aktuální potřeby verifikačního scénáře




Obrázek 3.1: Hlavní myšlenka portovatelných stimulů [4]

vyjádřeného grafem a tím dochází k výraznému snížení redundance. Portovatelné stimuly zachovávají pseudonáhodné generování stimulů s velkým snížením redundance díky řízení skrze graf. Díky absenci závislosti na implementačním jazyce a metodice verifikačního prostředí a výraznému snížení redundance, je možné použít pseudonáhodné generování stimulů i k testování na systémové úrovni.

Nový standard připravovaný organizací Accellera *"Portable Stimulus Standard"* (PSS) je novou specifikací jazyka pro vytváření portovatelných stimulů. Většina společností pro automatizaci elektronického návrhu hardwarových systémů má vlastní verzi jazyka pro portovatelné stimuly. Jednou z předních společností v této oblasti je Mentor se svým nástrojem Questa InFact, který je využíván v této práci. Dá se očekávat, že časem nástroj Questa InFact bude následovat PSS.

## 3.2 Popis nástroje InFact

Questa InFact je verifikační nástroj vytvářející sadu pravidel následně převedenou do grafové reprezentace, která je využívána algoritmy nástroje pro generování a/nebo směrování stimulů ve verifikačním prostředí, čímž vytváří různé verifikační scénáře během simulace. V této podkapitole budou postupně popsány klíčové komponenty nástroje pro vytvoření aplikace generující stimuly a způsob jakým zprostředkovává portovatelnost těchto aplikací. Tato podkapitola je postavena na [20].

Mezi klíčové komponenty potřebné pro vytvoření aplikace generující stimuly nástrojem Questa InFact patří soubor obsahující pravidla (angl. *rules file*, pro jednoduchost budeme dále využívat anglický název), graf (někdy také nazýván jako *test engine* nebo *run time graf*), strategie pokrytí (angl. *coverage strategy*) a testovací komponenta. Tyto jednotlivé komponenty budou nyní podrobněji rozebrány.

### 3.2.1 Soubor s pravidly

**Rules file** (soubor s koncovkou *.rules*) obsahuje sadu pravidel definovaných pomocí deklarativního jazyka. Celý soubor pravidel je převáděn do HVL nebo C++ jazyka podle toho, jaký jazyk byl zvolen pro verifikační prostředí, ke kterému bude výsledná aplikace připojena. Tato pravidla jsou vytvořena hierarchií výrazů, kde jsou listy následně vygenerovaného grafu v tomto souboru deklarovány jako akce. Pro větší přehlednost je možnost rozdělit tento hlavní soubor do více segmentových souborů mající koncovku *.rseg*. Tyto soubory mohou obsahovat stejné konstrukce jako hlavní soubor, kromě definice hlavní smyčky sekvence pravidel (grafu). Pro každou akci jsou vygenerovány metody v jazyce odpovídajícím verifikačnímu prostředí. Kód těchto metod lze podle potřeby upravovat, buď pomocí konstrukce **attributes** v souboru s pravidly, nebo přímo v generovaném souboru v jazyce verifikačního prostředí. Skrze pravidla je možné definovat různá chování, komplexní protokoly, specifické operace DUT, nebo je zde také možnost řídit softwarové operace v navrhovaných systémech obsahujících procesory pro společnou verifikaci hardwarové a softwarové části systému.

```
1  rule_graph comp_eng {
2      action init;
3
4      struct trans {
5          meta_action addr [unsigned 7:0];
6          meta_action data [unsigned 15:0];
7          meta_action size [1,2,4,8,16];
8          meta_action dir [enum READ, WRITE];
9
10         constraint mem_reg_c {
11             if (addr inside[0x10..0x1F]) {size <=8};
12             if (addr inside[0x40..0x5F]) {size inside[2,4]};
13         }
14
15         symbol mem_fields = size data;
16
17         trans = ((addr[0..7] dir[READ]) | (addr[8..255] dir)) mem_fields;
18     }
19     trans tr1, tr2;
20
21     interface do_tr(trans);
22     constraint diff_addrs_c {tr1.addr != tr2.addr}
23
24     comp_eng = init repeat {
25         do_tr(tr1)
26         do_tr(tr2)
27     };
28 }
```

Obrázek 3.2: Základní konstrukce **Rules file** [20]

Na obrázku 3.2 je možné vidět několik základních konstrukcí, které jsou zde blíže popsány.

- **rule_graph** - základní konstrukce, která je povinnou součástí rules file. Je uvozena klíčovým slovem **rule_graph** a vytváří deklaraci grafu. Tato konstrukce obaluje veškerý obsah tohoto souboru a kromě definování názvu grafu definuje také hlavní smyčku nebo sekvenci grafu (viz řádek 24 na obrázku 3.2 definující sekvenci pravidel nesoucí název celé konstrukce `rule_graph comp_eng`).

- **action** - klíčové slovo využívané při deklaraci akcí. Při průchodu grafem jsou volány metody, na které daná akce odkazuje (viz řádek 2 na obrázku 3.2 definující akci `init`). Tyto metody musejí být v jazyce odpovídajícím verifikačnímu prostředí. Jejich implementaci lze umístit přímo do souboru, který je generován v jazyce verifikačního prostředí nebo lze využít konstrukce *attributes* a umístit tuto implementaci do souboru k akcím.

- **struct** - deklaruje hierarchickou datovou strukturu, která obsahuje data, lokální omezení (omezení, angl. *constraints*, vztahující se pouze na obsah struktury) a informace o struktuře grafu (viz řádek 4 definující strukturu `trans` na obrázku 3.2). Struktura se může odkazovat jak na data definovaná uvnitř struktury tak i vně. Jednoduše je možné říci, že vytváří transakci s řízením generování jednotlivých stimulů, které obsahuje. Struktura je využívána pro definování komplexnějších chování, dává možnost definovat omezení mezi svými jednotlivými vytvořenými instancemi (viz řádek 22 na obrázku 3.2).

- **meta_action** - klíčové slovo pro deklaraci speciálního typu akcí. Tento typ akcí je využíván pro generování hodnoty typu integer z požadovaného rozsahu, kde výběr hodnoty provádí algoritmus nástroje Questa InFact (viz řádky 5 až 8 na obrázku 3.2). Hodnoty lze specifikovat výčtem jako je tomu např. na řádcích 7 a 8 na obrázku 3.2 nebo pomocí binární velikosti dané meta akce jako je tomu např. na řádcích 5 a 6 na obrázku 3.2. Meta akce jsou mapovány na proměnné uvnitř třídy, která představuje transakce ve verifikačním prostředí. Dalším typem akce jsou **meta_action_import**, které se liší od meta akcí pouze v tom, že hodnoty nejsou vybírány pomocí algoritmu nástroje, ale jsou přiváděny z verifikačního prostředí.

- **symbol** - slouží jako obálka, do které je možno umístit libovolnou grafovou konstrukci (posloupnost, smyčky, rozhodovaní etc.). Vytváří se pomocí něj hierarchie v grafu (viz řádek 15 na obrázku 3.2 definující symbol `mem_fields`, který je následně použit v definici sekvence pravidel uvnitř struktury na řádku 17 pro generování transakcí).

- **constraint** - definuje omezení, která se mohou vztahovat podle umístění a typu na různé části grafu. Jako například lokální omezení, která se vztahují pouze na data v dané instanci struktury nebo omezení globální vztahující se na celý graf (na obrázku 3.2 viz řádek 10 definující lokální omezení `mem_reg_c` a řádek 22 definující globální omezení `diff_addrs_c`).

- **interface** - deklaruje rozhraní pro specifikovanou strukturu (viz řádek 21 na obrázku 3.2) zprostředkovávající komunikaci s verifikačním prostředím.

- **sekvence pravidel** (angl. *rule_sequence*) - definuje pravidla pro řízení a generování stimulů při průchodu grafem pomocí deklarovaných akcí a symbolů (na obrázku 3.2 viz řádek 17 definující sekvenci pravidel pro instance struktury `trans` a řádek 24 definující sekvenci pravidel `comp_eng` tvořící celý výsledný graf).

Před dalším výkladem je ještě potřeba podrobněji vysvětlit konstrukce *constraint*, tedy definice omezení a také dosud nezmíněnou konstrukce *bins* a *bin_scheme*. Tento výklad je potřeba pro pochopení praktické části této práce.

### 3.2.2 Další konstrukce

V této podkapitole je uvedeno několik konstrukcí, které nebyly vysvětleny v podkapitole 3.2.1 a je zde rovněž blíže popsána konstrukce *constraint*. Všechny tyto konstrukce jsou využity při implementaci verifikačních scénářů portovatelných stimulů v této práci.

#### Constraint

Tato konstrukce definuje omezení, která se mohou vztahovat podle umístění a typu na různé části grafu. Nástroj Questa InFact rozlišuje tři typy *constraintů*:

- **Static** - tento typ constraintu musí být splněn pokaždé, když je vyhodnocována meta akce, ke které se vztahuje. V případě, že není specifikováno jinak, je defaultně nastaven typ constraintu jako static.
- **Dynamic** - tento typ constraintu vyžaduje klíčové slovo **dynamic** při jeho definici. Dynamický constraint je jako jediný typ constraintu vyobrazen v grafu. Musí být explicitně uveden v sekvenci pravidel, protože nabývá platnosti až v případě, že algoritmus nástroje přešel přes uzel, který jej představuje. Toto činí dynamický constraint závislý na procházené cestě.
- **Coverage** - tento typ constraintu vyžaduje klíčové slovo **coverage** při jeho definici. Coverage constraint se používá k definici omezení v konkrétní strategii pokrytí, ke které musí být rovněž přiřazen, aby nabyl platnosti.

Jak statické tak dynamické constrainty lze definovat ve struktuře i mimo ni. To také definuje způsob, jakým mohou být využity. Veškeré constrainty definované uvnitř struktury platí pouze v jejích instancích, lze tedy definovat různá omezení pro generování hodnot vztahující se na všechny vytvořené instance struktury, které představují transakce. Všechny typy constraintů lze kombinovat, je však nutné dávat pozor na to, aby se navzájem nevylučovaly (platí i pro vzájemné vyloučení constraintů univtř a vně struktury).

Například constraint na řádku 10 na obrázku 3.2 definuje omezení, které v závislosti na vygenerované hodnotě meta akce `addr` definuje omezení pro následné generování hodnoty meta akce `size`. Tento constraint nemá specifikován typ, jedná se tedy o statický constraint a nemusí být tudíž explicitně zadán do sekvence `trans`, která definuje jaké hodnoty a v jakém pořadí budou generovány pro danou transakci. Constraint na řádku 22 na obrázku 3.2 je rovněž statický, ale platí v celém grafu. Z obrázku lze vyčíst, že constrainty definované mimo strukturu se musí odkazovat na konkrétní instance dané struktury. V tomto příkladě je omezení definováno pro dvě transakce, od nichž je požadováno, aby nabývaly rozdílných hodnot meta akce `addr`.

Příklad dynamického constraintu lze vidět na obrázku 3.3 na řádcích 14 a 19, kde jsou definovány constrainty `read_addr` a `non_read_addr` spolu s klíčovým slovem `dynamic` uvnitř struktury. Pro uplatnění musí být tyto constrainty explicitně uvedeny v sekvenci pravidel `trans` definující transakci. Z obrázku 3.3 lze rovněž vyčíst, že tyto constrainty jsou každý součástí pouze jedné z větví a jimi definovaná omezení jsou tedy aplikována pouze při průchodu algoritmu nástroje danou větví. Část grafu odpovídající větvení vygenerovaného z tohoto zápisu je pak na obrázku 3.4. Tento příklad ve výsledku vytváří stejné transakce jako příklad v 3.2 s tím rozdílem, že tam bylo využito statického constraintu spolu s určením hodnot pro meta akce pomocí definování rozsahu přímo v sekvenci pravidel.

```
8   struct trans {
9       meta_action addr [unsigned 7:0];
10      meta_action data [unsigned 15:0];
11      meta_action size [1,2,4,8,16];
12      meta_action dir [enum READ, WRITE];
13
14      constraint read_addr dynamic{
15          addr inside [0..7] &&
16          dir == READ
17      }
18
19      constraint non_read_addr dynamic{
20          (
21              addr inside [0x10..0x1F] &&
22              size <= 8
23          )
24          |
25          (
26              addr inside [0x40..0x5F] &&
27              size inside [2,4]
28          )
29      }
30
31      trans =
32          read_addr
33          |
34          non_read_addr
35
36          addr
37          dir
38          size
39          data
40      ;
41  }
```

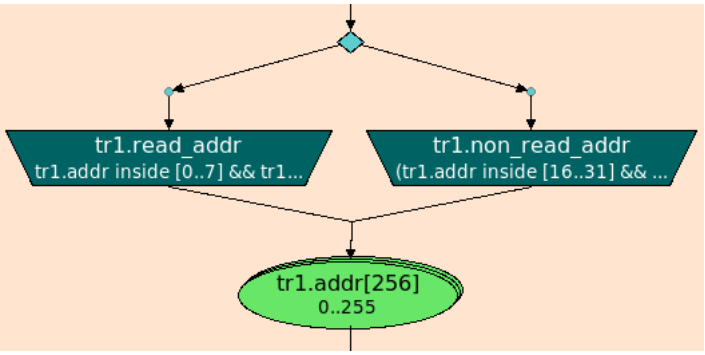
Obrázek 3.3: Implementace dynamického constraintu

### Bins a bin_scheme

Konstrukce *bins* definuje hodnoty nebo skupiny hodnot meta akcí, které mají být pokryty během simulace. V případě, že je meta akce s definovanými biny zahrnuta do strategie pokrytí, algoritmus nástroje cíleně generuje jednu náhodnou hodnotu pro každý definovaný bin. Hodnoty, které nejsou explicitně zahrnuty v konstrukci *bins* (pokud daná meta akce má tuto konstrukci definovánu) nejsou generovány algoritmem nástroje během simulace. Jsou rozlišovány dva typy binů, implictní a pojmenované. Biny implicitního typu jsou vytvářený bez názvů a uplatňují se vždy, pokud nejsou překryty pojmenovaným typem binů. Pro definování binů je vyhrazeno několik speciálních operátorů:

- Operátor pro dělení binů (/) - tento operátor dělí podmnožinu hodnot v binu na menší skupiny hodnot, jejichž počet odpovídá specifikovanému děliteli za tímto operátorem.
- Operátor pro velikost binů (:) - tento operátor dělí podmnožinu hodnot v binu na menší skupiny hodnot, kdy velikost těchto skupin odpovídá specifikované hodnotě za tímto operátorem.
- Divoká karta (angl. Wildcard) (*) - všechny hodnoty, které nespadají do žádného ze specifikovaných binů jsou umístěny do jednoho.

Příklad binu implicitního typu lze vidět na obrázku 3.5 na řádku 13. Tato konstrukce *bins* rozděluje celý rozsah hodnot definované meta akce `data` na několik dílčích skupin hodnot. Druhá z definovaných podmnožin v tomto binu odpovídající rozsahu hodnot 200..300 je dále rozdělena do menších skupin po 50 hodnotách pomocí operátoru (:). Místo tohoto operátoru lze také použít operátor (/), kdy pro dosažení stejného rozdělení této podmnožiny by byl dělitel roven 2. Je zde rovněž využit operátor (*), který v sobě zahrnuje všechny zbylé hodnoty, jenž nejsou součástí žádné z definovaných podmnožin tohoto binu.




Obrázek 3.4: Výsledný graf zobrazující dynamické constrainty

Konstrukce *bin_scheme* slouží jako obálka, do které je možné umístit libovolný počet pojmenovaných i implicitních binů. Rovněž je tato konstrukce nutná v případě použití pojmenovaných binů, které sice mohou být definovány mimo tuto konstrukci, ale musejí být uvedeny v nějaké konstrukci *bin_scheme*, aby nabyly platnosti. Příklad konstrukce *bin_scheme* lze vidět na obrázku 3.5 na řádku 26. Zde tato konstrukce v sobě uchovává odkaz na pojmenovaný bin `sz_bns_1`, který rozděluje rozsah hodnot meta akce `size` do tří podmnožin. Některé hodnoty nejsou v tomto binu zahrnuty v žádné z podmnožin, například hodnota 1023, a nedojde tedy k jejich vygenerování v průběhu simulace. Rovněž je v této konstrukci také definován bin implicitního typu, jenž rozděluje rozsah hodnot meta akce `addr` do tří binů, kdy dva z nich jsou jedna konkrétní hodnota a třetí je podmnožina hodnot z rozsahu hodnot definovaných pro tuto meta akci.

```
8   struct trans {
9       meta_action size [0..4095];
10      meta_action addr [0..255];
11      meta_action data [0..1024];
12
13      bins data
14          [0]
15          [200..300]:50
16          [572..1024]
17          [*]
18          ;
19
20      bins size sz_bns_1// named bin definition
21          [1..1022] // small packets
22          [1025..2046] // mid packets
23          [2049..4094] // large packets
24          ;
25
26      bin_scheme sz_addr_sch {
27          size sz_bns_1; // reference to defined named bin
28          addr [0][1..254][255];
29      };
30  }
```

Obrázek 3.5: Ukázka konstrukce bin_scheme a různých binů

### Metody grafu

Před dalším výkladem je třeba vysvětlit co to jsou *test engine metody* (výraz test engine je v tomto případě používanější než graf). Všechny tyto metody je možné použít pouze v generovaném souboru obsahující podobu souboru pravidel v jazyce odpovídajícím verifikačnímu

prostředí. Využívají se pro ovlivnění průchodu grafem během simulace. Jsou nadefinovány celkem 4 test engine metody:

- Vynutit další akci (angl. *force next action*) - alternativy jsou vybírány algoritmem nástroje podle strategie pokrytí nebo pravděpodobnostních tagů (probráno v následující podkapitole). Tato metoda poskytuje možnost výběru alternativy na základě stavu DUT nebo verifikačního prostředí. Lze ji volat z jakékoliv akce uvnitř definované sekvence pravidel pro explicitní výběr následující akce.
- Přeskočit na akci (angl. *skip to action*) - metoda poskytuje způsob jak skočit bezpodmínečně na jinou akci. Podobá se to konstrukci *"goto"* používané při programování softwaru.
- Zastavení smyčky po (angl. *stop loop expansion after*) - metoda využívána pro ukončení smyčky. Pomocí parametrů lze určit kolikrát se má daná smyčka vykonat (určeno prvním parametrem) a z kolika smyček se má vynořit (určeno druhým parametrem, číslováno od nuly, která představuje aktuální prováděnou smyčku) v případě více do sebe zanořených smyček.
- Nastavení pravděpodobnosti tagu (angl. *set tag probability*) - metoda pro nastavování vah odpovídajících tagů pravděpodobnosti. Pokud v nějakém bodě grafu mají všechny větve grafu dohromady váhu větší než jedna, jsou tyto váhy normalizovány do rozsahu 0 až 1 a jakákoliv větev bez přiřazené váhy nebo pravděpodobnostního tagu má automaticky přiřazenu nulovou váhu a je tudíž odpojena. Pokud mají naopak všechny větve dohromady v nějakém bodě váhu menší než jedna, je zbytek rozdělen mezi zbývající větve.

### Pravděpodobnost

V případě různých alternativ pravidel, v grafu vyobrazených jako větvení, volí algoritmus nástroje Questa InFact variantu na základě strategie pokrytí. Kromě řízení procházení grafu na základě strategie pokrytí je zde také možnost využít *pravděpodobností tagy*. Tagy mohou rozhodovat o volbě cesty na základě pravděpodobnosti nebo relativní váhy. Jsou užitečné v následujících aplikacích:

- Soustředění simulace na zajímavější alternativu (cestu).
- Zaměření problémové oblasti návrhu jako první pro maximalizaci efektivity simulace a nalezení chyb rychleji.
- Vyřazení funkcionality verifikačního prostředí, které doposud nebylo implementováno nebo laděno.
- Opakovaně provádět určité alternativy a vyřazovat druhé pro nastavení DUT na určitou konfiguraci. Jako příklad lze uvést alternativu na plnění a vyprázdnění FIFO fronty, jako prerekvizitu pro vykonání operací závislých na stavu FIFO fronty.

Tagy jsou přiřazovány použitím klíčového slova *tag* následovaného jeho názvem. Na obrázku 3.6 je možné vidět vytvoření dvou různých tagů na řádcích 15 a 21.

Takto definované tagy nemají žádný účinek. Nejdříve je potřeba využít jednu z test engine metod, konkrétně *set_tag_probability* (viz předchozí podkapitola). Na obrázku 3.7

```
8   struct trans {
9       meta_action addr [unsigned 7:0];
10      meta_action data [unsigned 15:0];
11      meta_action size [1,2,4,8,16];
12      meta_action dir [enum READ, WRITE];
13
14      trans =
15          tag prob_1
16          (
17              dir[READ, WRITE]
18              addr
19          )
20          |
21          tag prob_15
22          (
23              dir[WRITE]
24              addr[8..255]
25          )
26          size
27          data
28      ;
29  }
```

Obrázek 3.6: Ukázka definice pravděpodobnosti v souboru pravidel

jsou přiřazeny váhy pravděpodobnostním tagům definovaným výše, kde `m_te` je název test enginu v tomto vygenerovaném souboru. Součet přiřazených vah je vyšší než jedna a dojde tedy k jejich normalizaci.

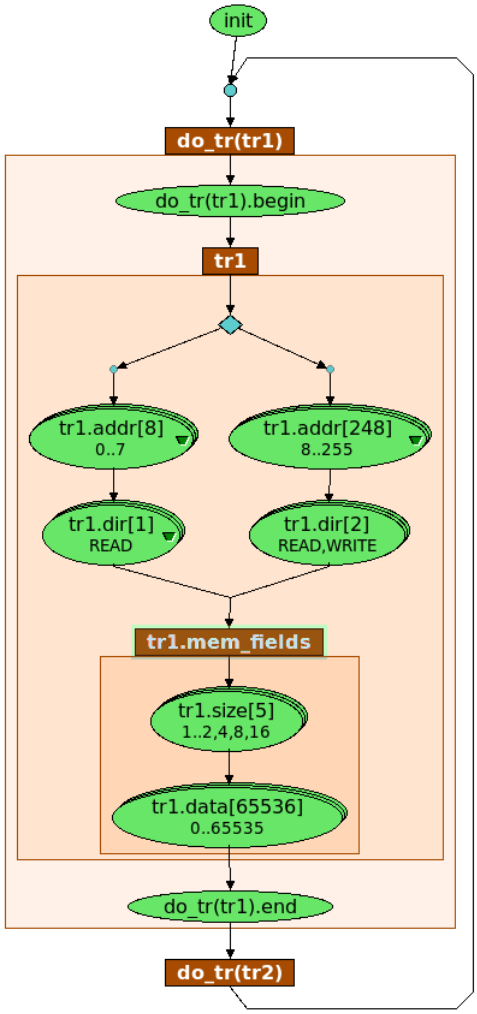
```
154 //<action::_init::()>
155     //*********************************************************************
156     //* Action task action_init
157     //*********************************************************************
158    virtual task action_init();
159     m_te.set_tag_probability("prob_1", 1);
160     m_te.set_tag_probability("prob_15", 15);
161     endtask
162 //</action::_init::()>
```

Obrázek 3.7: Přiřazení vah pravděpodobnostním tagům

### 3.2.3 Graf

**Graf** je generován a kompilován automaticky jako binární reprezentace rules file. Každá akce definovaná v rules file je vyobrazena jako uzel grafu. Během simulace algoritmy nástroje procházejí grafem a rozhodují pomocí něj o výběru hodnot generovaných stimulů.

Na obrázku 3.8 je možné vidět graf vytvořený na základě pravidel definovaných na obrázku 3.2. Na graf lze pohlížet jako na vývojový diagram. Například na pravé větvi je nejdříve zvolena adresa z rozmezí hodnot 8 až 255 a následně je voleno, zda dojde ke čtení či zápisu z této adresy atd. Je možné vidět, že celý graf odpovídá sekvenci pravidel přiřazené názvu grafu (viz řádek 24 na obrázku 3.2). Na obrázku je dále rozbalena instance struktury `tr1`, kde je ukázáno větvení definované jako sekvence pravidel přiřazených názvu struktury (viz řádek 17 na obrázku 3.2). Je zde rovněž rozbalen symbol, který v sobě ukrývá meta akce `size` a `data`. Všechny meta akce v grafu mají ve svých uzlech také vyobrazeny rozsahy hodnot, kterých mohou nabývat. Celá část grafu odpovídající struktuře je dále ohraničená rozhraním `do_tr`, které slouží pro zasílání generovaných stimulů do verifikačního prostředí.

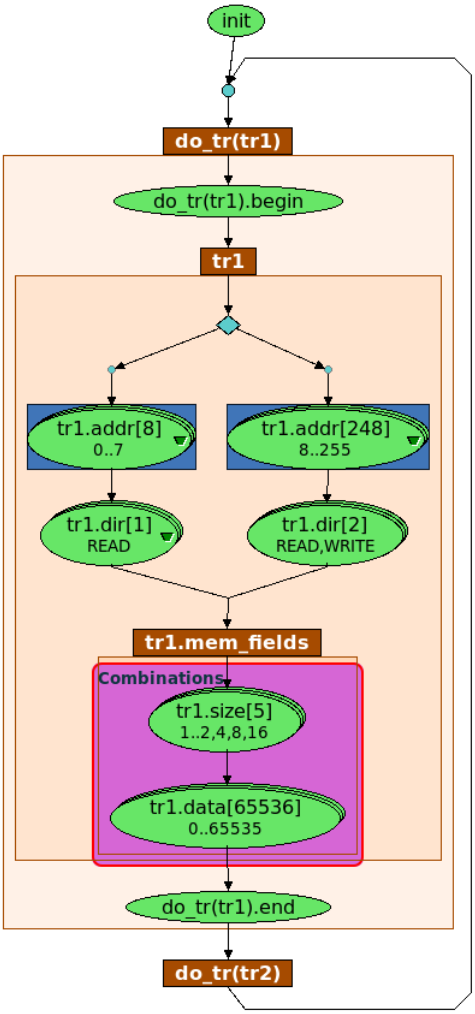



Obrázek 3.8: Graf vygenerovaný z pravidel definovaných na obrázku 3.2

### 3.2.4 Strategie pokrytí

**Strategie pokrytí** je generována přímo z grafu a slouží k zaměření algoritmů nástroje na určité vlastnosti DUT. Pokud je v grafu hlavní smyčka a není definována strategie pokrytí nelze mít jako podmínku pro ukončení smyčky splnění pokrytí, protože by nedošlo k ukončení simulace.

Na obrázku 3.9 lze vidět pokrytí akce a cesty. Pokrytí akce (angl. *Action Coverage*) sděluje algoritmu nástroje, aby byly postupně vygenerovány všechny její možné hodnoty definované při její deklaraci (na obrázku 3.9 jsou to uzly označené `tr1.addr`). Pokrytí cesty (angl. *Path Coverage*), na grafu pojmenované jako `Combinations`, sděluje algoritmu nástroje, aby provedl generování všech kombinací hodnot daných akcí zahrnutých v této cestě. V tomto konkrétním příkladě budou vygenerovány všechny možné kombinace hodnot akcí `size` a `data`. Pokud by na grafu byla nějaká delší cesta a nebylo by vyžadováno do kombinací zahrnout některou z akcí, je možné ji označit jako ignorovanou. Hodnoty této akce budou nadále generovány, ale algoritmus nebude brát v potaz zda byly již vygenerovány kombinace se všemi jejími možnými hodnotami. Pokud byla vytvořena strategie pokrytí, ale nejsou definována žádná pokrytí akce ani cesty a je zachována podmínka pro ukončení




Obrázek 3.9: Strategie pokrytí pro výše uvedený graf

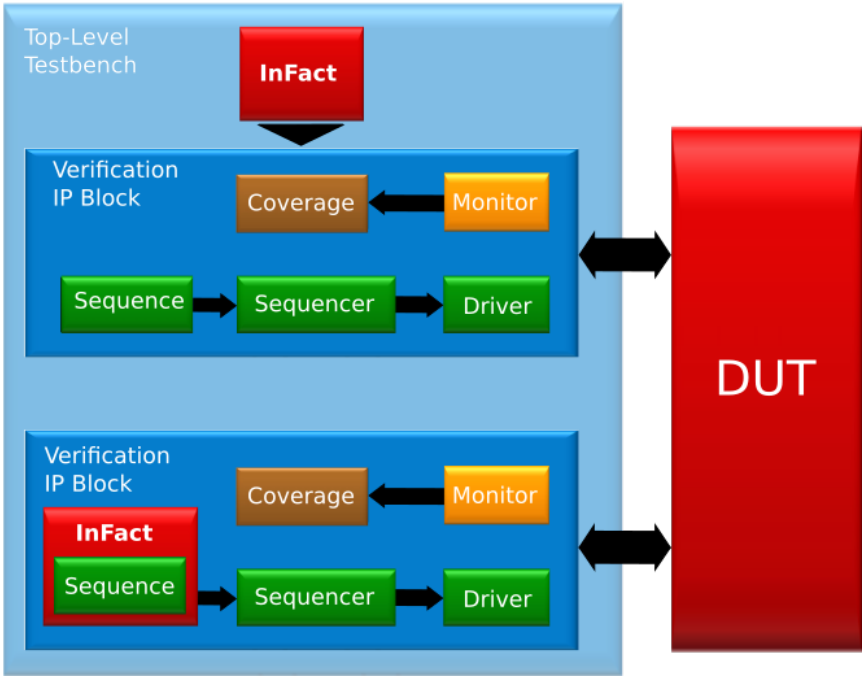
testování na základě splnění pokrytí, dojde k okamžitému ukončení simulace. V případě že nejsou definovány hodnoty pomocí konstrukcí *bins* nebo *bin_scheme* určující jakých hodnot je požadováno, aby akce zahrnuté do strategie pokrytí nabývaly, tak algoritmus nástroje Questa InFact postupně generuje veškeré možné hodnoty těchto akcí, které jsou definovány při jejich deklaraci, kdy například pro meta akci `data` by bylo vygenerováno všech 65 536 hodnot. Pokrytí se netýká všech instancí struktury, ale pouze konkrétních uzlů na kterých je definováno (na obrázku 3.9 jsou všechna pokrytí definována na uzlech instance `tr1` struktury). Je třeba si dát pozor, aby bylo pokrytí definováno pro účely testování přesně a nedošlo k jeho předčasnému ukončení.

### 3.2.5 Testovací komponenty

Nástroj generuje implementaci pro vytvořenou aplikaci jako samostatnou testovací komponentu verifikačního prostředí a následně tato komponenta běží ve spojení se simulátorem při funkční verifikaci. Testovací komponenta je obálka vytvořená podle vybrané metodiky a jazyka verifikačního prostředí, ke kterému je následně připojena. Toto připojení testovací

komponenty k verifikačnímu prostředí vyžaduje malou nebo dokonce žádnou modifikaci verifikačního prostředí.

Testovací komponenta je k verifikačnímu prostředí zapojena různými způsoby v závislosti na aplikaci, tedy na požadavcích verifikačního inženýra na testovací komponentu. Testovací komponenta InFact může kontrolovat generování stimulů, nebo může také řídit konfiguraci verifikačního prostředí, nebo může provádět řízení jak generování stimulů tak řídit konfiguraci verifikačního prostředí. Podle těchto požadavků je také testovací komponenta připojena přímo k strukturní komponentě verifikačního prostředí sequencer a/nebo je umístěna do top modulu jak je ukázáno na obrázku 3.10, kde je vyobrazena kombinace řízení konfigurace i generování stimulů testovací komponentou.




Obrázek 3.10: Zapojení testovací komponenty do verifikačního prostředí 3.2

### 3.2.6 Portovatelnost

Implementace sady pravidel definujících graf je naprosto nezávislá na jazyce a metodice verifikačního prostředí. Tato vlastnost umožňuje využívat tato pravidla v různých verifikačních prostředích bez potřeby provádět na nich nějaké změny. Je třeba pouze vybrat správný jazyk a metodiku, podle které bude vytvořena testovací komponenta a provést její připojení k požadovanému verifikačním prostředí [5]. Díky absenci závislosti na implementačním jazyce a metodice verifikačního prostředí a výraznému snížení redundance, je možné použít pseudonáhodné generování stimulů i k verifikaci na systémové úrovni, kde nástroj rovněž umožňuje provést společnou verifikaci hardwarové a softwarové části.

Testovací komponenty lze také používat jako součást jiných testovacích komponent, které byly vytvořeny pro vyšší úroveň abstrakce. Je tedy možné testovací komponenty vytvořené pro IP bloky *"spojit"* v testovací komponentě, která je určena pro testování subsystému obsahujícím tyto IP bloky a totéž platí ve vztahu testovacích komponent určených pro testování systému jako celku a testovacích komponent subsystémů.

# Kapitola 4

# RISC-V

RISC-V je volně dostupná instrukční sada (angl. *Instruction Set Architecture*, ISA) postavená na principech instrukční sady RISC (zkratka z angl. *Reduced Instruction Set Computing*). Tento projekt započal v roce 2010 na Kalifornské univerzitě v Berkeley v USA a byl původně určen pro podporu výuky a výzkumu počítačových architektur. ISA RISC-V se však dočkala nebývalého přijetí v akademickém i průmyslovém světě a byla proto založena nezisková nadace *RISC-V Foundation* pro její ochranu a podporu jakožto nového průmyslového standardu. Úvodní část této kapitoly čerpá z [24] a [25].

Cílem RISC-V je ISA, kterou bude možné implementovat přímo jako hardware a nebude zaměřená na jeden typ mikroarchitektury nebo implementační technologie (ASIC, FPGA). RISC-V podporuje vysokou paralelizaci více jader, což umožňuje vytvářet hierarchickou strukturu i malým systémům na čipu v podobě modulů. Aktuálně jsou definovány tři privilegované režimy:

- Strojový (angl. *Machine*, M) - má nejvyšší privilegia a jako jediný je povinný pro všechny hardwarové platformy RISC-V. Kód běžící v tomto režimu má nízko-úrovňový přístup k implementaci stroje a může být použit pro správu bezpečného provádění prostředí na RISC-V.
- Uživatelský (angl. *User*, U) - je určen pro použití běžných aplikací a operačních systémů.
- Dozorčí (angl. *Supervisor*, S) - stejně jako uživatelský režim a navíc také zde byla umístěna podpora virtualizace RISC-V ve formě rozšíření. Toto rozšíření nahradilo privilegovaný virtualizační (angl. *Hypervisor*, H) režim, jenž byl odstraněn.

Privilegované režimy lze v implementacích RISC-V kombinovat jak je znázorněno v tabulce 4.1.

Tabulka 4.1: Kombinace privilegovaných režimů

| **Počet úrovní** | **Podporované režimy** | **Zamýšlené použití** |
|---|---|---|
| 1 | M | Jednoduché vestavěné systémy |
| 2 | M, U | Zabezpečené vestavěné systémy |
| 3 | M, S, U | Systémy s Unixovým operačním systémem |

Na následujících stránkách je blíže popsána ISA RISC-V a její rozšíření. Jsou zde také zmíněny moduly mikroarchitektury vyvinuté na základě architektury RISC-V od společnosti Codasip, pro které budou navrhovány a následně implementovány portovatelné verifikační scénáře.

## 4.1 ISA

RISC-V ISA je tvořena instrukcemi pro celočíselné operace jako základem, který musí být přítomen v každé implementaci, a volitelnými rozšířeními tohoto základu. Celočíselný základ ISA je velice podobný tomu v raných ISA RISC procesorech až na absenci opožděných skoků (angl. *branch delay slots*) a podporu proměnné délky instrukcí. Tato podkapitola čerpá z [25].

RISC-V obsahuje více variant instrukčních sad charakterizovaných podle šířky celočíselných registrů a odpovídající velikosti adresového prostoru. RISC-V podporuje 32 a 64 bitové adresové prostory a je připraven i na verzi pro 128 bitové adresové prostory, která může být v budoucnu potřeba. Jednotlivé části ISA RISC-V jsou označovány **RV32I**, kde **RV** je zkratka pro RISC-V, **32** je bitová šířka celočíselných registrů (32, 64, 128) a písmeno **"I"** značí, že se jedná o instrukce pro celočíselné operace (celá čísla angl. *Integer*, I). Některé z procesorů řady Berkelium od společnosti Codasip, na které je tato práce zaměřena, využívají i instrukční sady označované **RV32E**, jenž je redukovanou verzí **RV32I** navržené pro vestavěné systémy. Hlavními rozdíly jsou redukovaný počet celočíselných registrů a odstranění čitačů povinných v **RV32I**. Označení pro jednotlivá standardní rozšíření jsou uvedena dále v textu. Instrukce pro celočíselné operace tvořící základ ISA RISC-V jsou instrukce výpočetní (sčítání, odčítání, logické operace a porovnávání), čtecí, zápisové a pro řízení toku (skoky).

### 4.1.1 Volitelná rozšíření

Implementace procesoru RISC-V musí podporovat celočíselné operace a může podporovat jedno či více rozšíření. Tato volitelná rozšíření jsou dělena na **standardní**, která jsou obecně užitečná a navržená tak, aby nebyla v konfliktu s ostatními standardními rozšířeními, a na **nestandardní**, která mohou být vysoce specializovaná a mohou být v konfliktu jak se standardními tak nestandardními rozšířeními. Je zde předpoklad vývoje mnoha různých nestandardních rozšíření, z nichž některá se mohou po čase zařadit mezi rozšíření standardní. Standardní rozšíření se dále dělí na rozšíření pro obecné použití a rozšíření na uživatelské úrovni. Všechna rozšíření obsahující instrukce pro práci s desetinnými čísly jsou v souladu s aritmetickým standardem IEEE 754-2008.

#### Standardní rozšíření pro obecné použití

Standardní rozšíření pro obecné použití s označením **"M"** (z angl. *Multiplication*) přidává instrukce pro násobení a dělení hodnot uložených v celočíselných registrech. Rozšíření označované jako **"A"** (z angl. *Atomic*) přidává instrukce atomického čtení, modifikace a zápisu do paměti pro mezi-procesorovou synchronizaci. Rozšíření s označením **"F"** (z angl. *Float*) přidává registry pro čísla s plovoucí řádovou čárkou a výpočetní, čtecí a zápisové instrukce s jednoduchou přesností (angl. *single-precision*). Standardní rozšíření nesoucí označení **"D"** (z angl. *Double*) rozšiřuje registry pro čísla s plovoucí řádovou čárkou a přidává výpočetní, čtecí a zápisové instrukce s dvojitou přesností (angl. *double-precision*). Spolu se základ-

ními instrukcemi pro celočíselné operace (výsledné označení této kombinace je **"IMAFD"**) poskytují tato standardní rozšíření skalární instrukční sadu pro obecné použití nesoucí zkrácené označení **"G"** z angl. *Global-purpose*. Celkové výsledné označení je **RV32G**, **RV64G** nebo **RV128G** v závislosti na bitové šířce celočíselných registrů.

### Standardní rozšíření na uživatelské úrovni

Standardní rozšíření na uživatelské úrovni s označením **"Q"** (z angl. *Quad*) přidává instrukce s čtyřnásobnou přesností pro čísla s pohyblivou řádovou čárkou. Toto rozšíření vyžaduje implementaci **RV64IFD** a zvětšení registrů pro čísla s pohyblivou řádovou čárkou až na 128 bitů, což umožňuje v registrech uchovávat desetinná čísla s jednoduchou, dvojitou i čtyřnásobnou přesností.

Rozšíření nesoucí označení **"C"** (z angl. *Compressed*) je určeno pro redukci velikosti kódu, což realizuje přidáním 16 bitového kódování instrukcí pro běžné operace. Typicky je možné těmito instrukcemi nahradit 50% - 60% instrukcí RISC-V v programu a tím dosáhnout zmenšení velikosti výsledného kódu o 25% - 30%.

Kromě dvou výše uvedených rozšíření existují i méně definovaná, která slouží spíše jako návrhy do budoucna mající rezervovaná označení. Rozšíření s označením **"L"** je určeno pro podporu dekadické aritmetiky nad čísly s plovoucí řádovou čárkou podle definice v aritmetickém standardu IEEE 754-2008. Rozšíření pro instrukce bitové manipulace mají rezervováno označení **"B"** (z angl. *Bit*) a zahrnují bitové operace vkládání a extrakci, testování, rotaci a posuv bitových polí, bitové a bytové permutace. Další zamýšlené rozšíření je určeno pro podporu dynamicky překládaných jazyků a nese označení **"J"**. Rozšíření nesoucí označení **"T"** (z angl. *Transactional*) je aktuální myšlenka v rámci stále probíhajících debat o nejlepším způsobu podpory atomických operací zahrnujících více adres. Myšlenkou tohoto řešení je zahrnout malý, kapacitně omezený transakční paměťový buffer. Označení **"P"** (z angl. *Packed SIMD*) je rezervováno pro paralelní zpracování více dat jedinou instrukcí s využitím registrů určených pro čísla s plovoucí řádovou čárkou. Návrh rozšíření nesoucí označení **"V"** (z angl. *Vector*) má za cíl podporovat konfigurovatelnost vektorových jednotek pro umožnění běhu jednoho binárního kódu na různých hardwarových implementacích lišících se ve velikostech úložného prostoru vektorových jednotek a paralelismu datových cest. Označení **"N"** bylo rezervováno pro návrh rozšíření realizující přerušení a zpracování výjimek.

### Nestandarní rozšíření

Nestandardní rozšíření nejsou značena jedním písmenem, ale celým názvem. Před název je umístěno velké písmeno “X“ a je také možné přidat za název číslo verze rozšíření. Při více nestandardních rozšířeních dochází ve značení k jejich rozdělení pomocí znaku podtržítka ("RV64GXargle_Xbargle").

Jako příklad nestandardního rozšíření je možno uvést rozšíření Hwacha. Architektura postavená na tomto rozšíření je zaměřena na využití vysokého stupně oddělení přístupu k datům vektoru a zpracování vektorů. Hwacha architektura je postavena tak, aby poskytovala vysoký výkon při nízké spotřebě energie pro širokou škálu aplikací. Na rozdíl od ostatních architektur zaměřených na vektory se snaží Hwacha tlačit limity oddělených přístupů skrze sestavovací programovací model pro vektorové načítaní (angl. *vector-fetch*), který skládá všechny vektorové instrukce do samostatného vektor načítajícího bloku (angl. *vector-fetch block*) [19].

## 4.2 Berkelium

Codix Berkelium je procesorová řada společnosti Codasip implementující architekturu RISC-V. Rozličná nastavení, plná konfigurovatelnost a rozšiřitelnost poskytuje výhodu oproti tradičním pevně nakonfigurovaným IP jádrům. Procesory této řady implementují různé podmnožiny architektury RISC-V skrze konfigurovatelné moduly. Díky modulům implementujícím různá rozšíření architektury RISC-V je poskytnuta procesory Berkelium vysoká míra flexibility. Postupně jsou v této podkapitole blíže popsány některé prvky procesorů této řady, zejména Codix Bk1 (Berkelium s automatovou řídící logikou), Codix Bk3 (Berkelium s třístupňovým zřetězením zpracování instrukcí), Codix Bk5 (Berkelium s pětistupňovým zřetězením zpracování instrukcí). Tato kapitola čerpá z [6], [7] a [8].

### 4.2.1 Přerušení

Požadavek na přerušení zůstává na aktivní úrovni dokud mu není zasláno oznámení o jeho zpracování od obsluhy přerušení. Typická doba trvání přerušení pro procesor Codix Bk3 a Codix Bk5 je jeden hodinový cyklus. Doba trvání přerušení pro procesor Codix Bk1 závisí na konfiguraci jádra a podle ní jsou doby trvání přerušení různé:

- Základní doba trvání přerušení je až 4 hodinové cykly.
- V případě poskytování rozšíření pro redukci velikosti kódu nesoucí označení **"C"** (z angl. *Compressed*) procesorem, je doba přerušení zvýšena o jeden hodinový cyklus.
- V případě poskytování rozšíření pro násobení a dělení nesoucí označení **"M"** (z angl. *Multiplication*) procesorem, je doba přerušení zvýšena až o 31 hodinových cyklů.

Procesorové jádro musí před vstupem do obsluhy přerušení počkat na dokončení přenosu dat po sběrnici pro případ, že by se jednalo o výjimku na sběrnici. Procesory mají prioritu přerušení v tomto sestupném pořadí:

- Externí přerušení
- Přerušení od časovače
- Softwarové přerušení

### 4.2.2 Řízení spotřeby

Pro řízení spotřeby je určena speciální jednotka (angl. *Power Management Unit*, PMU) obsahující registr uchovávající posloupnost kroků při probouzení procesoru z režimu spánku. PMU je zodpovědná za dodržení správné sekvence jak při probouzení, tak i při resetu. Rovněž poskytuje signály pro řízení režimu spotřeby procesorového jádra. PMU dostane při uspání signál a podle nastaveného režimu spánku provede patřičné operace. Procesory poskytují tyto režimy spánku:

- **Stop mode** - v tomto režimu spánku jádro vstoupí do tzv. *internal sleep mode* a čeká na externí událost. Jádro je napájeno a stále dostává hodinový signál, ale nevykonává žádné instrukce. V tomto módu není od PMU vyžadováno nic.
- **Clock gating mode** - v tomto režimu spánku je od PMU vyžadováno zastavení hodin.

- **Power retention mode** - v tomto režimu spánku je od PMU vyžadováno vypnutí napájení jádra, a zároveň ponechání napájení pro retenční registry pro zachování v nich uložených hodnot.

Při uspání procesoru, jádro vždy vstoupí do nastaveného režimu spánku. Běžící proces však potřebuje rovněž definovat PMU co se stane s prostředím jádra. Pro tento účel je předem definován pouze jeden režim, který prostředí procesorového jádra nechává v aktuálním nastavení spotřeby a tedy pouze jádro přechází do režimu nízké spotřeby, ostatní režimy jsou definovány při implementaci.

### 4.2.3 L1 cache

Procesory (kromě procesoru Codix Bk1) podporují datovou a instrukční paměť typu cache úrovně L1. Paměť cache je navržena jako synchronní a lze pro tuto paměť konfigurovat následující:

- Velikost paměti cache
- Velikost cache řádku (angl. *cache line*) - tyto řádky jsou nejmenší bloky datových slov v paměti cache.
- Algoritmus paměti cache (angl. *cache replacement algorithm* nebo *cache replacement policy*) - algoritmus zodpovídá za správu uložených dat v cache paměti (např. kam ukládat nová data, kdy jsou data neplatná atd.).
- Počet cest N - paměť cache je rozdělena do více úseků, což umožňuje mapování řádku paměti do více cache řádků (N asociativní paměť cache). Hodnota parametru N udává počet míst, na která jsou data pro daný index uložena (index je část nastavené adresy při ukládání/čtení dat).
- Oblasti neukládající se do cache paměti

Instrukční cache paměť provádí přednačítání následujícího cache řádku v čase, kdy jsou čtena poslední data z aktuální cache paměti.

Komunikace mezi pamětí cache a procesorem je postavena na protokolu lokální sběrnice od společnosti Codasip (angl. *Codasip Local Bus*, CLB) viz 4.2.4.

### 4.2.4 Sběrnice

Pro komunikaci s procesory Berkelium slouží lokální sběrnice od společnosti Codasip (angl. *Codasip Local Bus*, CLB). Jedná se o vysoko výkonovou synchronní sběrnici, která zajišťuje komunikaci s operační pamětí, pamětí typu cache a jinými širokopásmovými periferními zařízeními. CLB podporuje bus mastering s jediným masterem. Šířka datové sběrnice je až 1024 bitů a šířka adresové sběrnice je až 64 bitů. Operace nad komunikačním protokolem CLB trvá vždy minimálně dva hodinové cykly. V prvním hodinovém cyklu dochází k vystavení adresy na sběrnici a v druhém následně dochází ke zpracování dat.

Kromě sběrnice CLB je umožněno využívat různé další sběrnice odpovídající standardu AMBA (angl. *Advanced Microcontroller Bus Architecture*). AMBA je otevřený standard pro specifikaci vnitřního propojení na čipu a správu funkčních bloků u návrhů systémů na čipu (angl. *System on a Chip*, SoC). AMBA byla poprvé představena v roce 1996 společností

ARM a první sběrnice odpovídající tomuto standardu byly ASB (angl. *Advanced System Bus*) a APB (angl. *Advanced Peripheral Bus*).

Procesory Codix Berkelium poskytují možnost připojení k procesoru pomocí sběrnic **AHB Lite** a **AXI4 Lite** odpovídající standardu AMBA. Pro komunikaci po těchto sběrnicích se využívá propojení přes tzv. můstek (angl. *bridge*), kde dochází k překladu ze signálů CLB rozhraní na rozhraní odpovídající sběrnice. Takový způsob připojování nových sběrnic vyžaduje pouze implementaci odpovídajícího můstku při zachování téhož rozhraní na procesoru. Můstek je typu master-master, jde tedy pouze o transformaci základního rozhraní (CLB) na rozhraní jiné sběrnice (AHB Lite a AXI4 Lite), kterou lze tímto způsobem použít.

V případě AHB Lite můstku jsou dva možné způsoby jeho využití, v obou je Codix Berkelium procesor masterem. V prvním případě lze připojit skrze můstek tzv. *AHB Lite control* komponentu pro možnost připojení více zařízení typu *AHB Lite slave*. V druhém případě lze zařízení typu *AHB Lite slave* připojit přímo, pouze je nutné, aby můstek obsahoval dva signály navíc.

### 4.2.5 Vykonávání instrukcí

Procesor Codix Bk1 vykonává současně pouze jednu instrukci. Nicméně je zde překrytí jednoho hodinového cyklu u po sobě jdoucích instrukcí, protože načítaní nové instrukce začíná současně s dokončením aktuální instrukce. Počet cyklů na instrukci záleží na typu vykonávané instrukce:

- Dokončení instrukcí násobení a dělení trvá 36 hodinových cyklů.
- Dokončení instrukcí load a store trvá 5 hodinových cyklů v případě že zpoždění (latence) paměti je rovna 1.
- Dokončení všech ostatních instrukcí trvá 4 hodinové cykly.

Procesor Codix Bk3 obsahuje třístupňové zřetězené zpracování instrukcí (viz obrázek 4.1), které je určené pro internet věcí (angl. *Internet of Things*, IoT) aplikace s velmi nízkou spotřebou, jenž nechtějí obětovat výkon. IoT je síť fyzických zařízení, vozidel, domácích spotřebičů a dalších, která jsou vybavena elektronikou, softwarem, senzory, pohyblivými částmi a síťovým připojením, které jim umožňuje jejich vzájemné propojení a výměnu dat.


Obrázek 4.1: Zřetězené zpracování instrukcí procesoru Codix Bk3 [8]

Procesor Codix Bk5 obsahuje pětistupňové zřetězené zpracování instrukcí (viz obrázek 4.2) umožňující jeho běh na vyšších frekvencích a integraci více funkcí bez obětování rychlosti.


Obrázek 4.2: Zřetězené zpracování instrukcí procesoru Codix Bk5 [8]

### 4.2.6 Podporovaná rozšíření RISC-V

Kromě rozšíření podporují jednotlivé procesory i privilegované režimy. Procesory Codix Bk1 a Codix Bk3 v sobě zahrnují částečnou podporu privilegovaných režimů pro podporu bezpečnosti operačních systémů běžících na vestavěných systémech. Procesor Codix Bk5 v sobě zahrnuje podporu privilegovaných režimů umožňující běh např. operačního systému FreeRTOS. V tabulce 4.2 je ukázán stručný přehled podporovaných rozšíření jednotlivými procesory řady Berkelium od společnosti Codasip.

Tabulka 4.2: Rozšíření RISC-V podporované jednotlivým procesory Berkelium

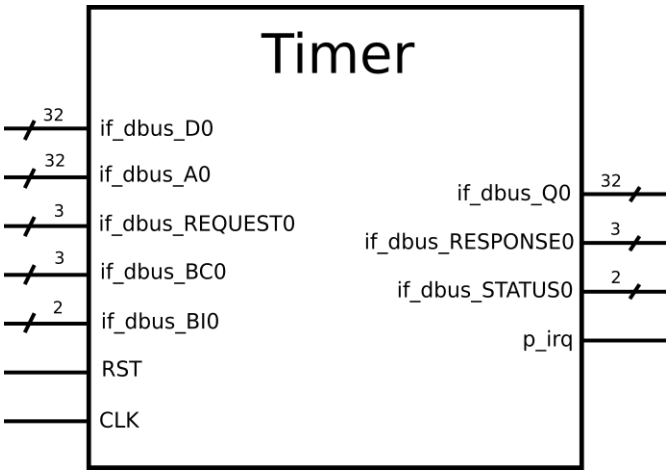
| **Rozšíření RISC-V** | **Značení** | **Codix Bk1** | **Codix Bk3** | **Codix Bk5** |
|---|---|---|---|---|
| Celočíselné operace | I | Ne | Ano | Ano |
| Redukovaná verze I pro vestavěné systémy | E | Ano | Ano | Ano |
| Násobení/dělení | M | Ano (sériově) | Ano (paralelně/sériově) | Ano (paralelně/sériově) |
| Redukce velikosti kódu | C | Ano | Ano | Ne |
| Floating point operace | F | Ne | Ne | Ano |
| Floating point operace s dvojitou přesností | D | Ne | Ne | Ano |
| Privilegované režimy | | Ne | Částečně | Částečně |
| Atomické čtení/zápis a modifikace paměti | A | Ne | Ne | V závislosti na sběrnici |

## 4.3 Moduly

V této části kapitoly jsou postupně popsány moduly procesoru Berkelium implementujícím architekturu RISC-V od společnosti Codasip, pro něž jsou vytvořeny portovatelné verifikační scénáře za pomoci nástroje Questa InFact.

### 4.3.1 Časovač

Jednoduchý časovač vytvořený jako modul procesoru Berkelium je navržen s ohledem na dokumentaci architektury RISC-V [24], podle které je procesor implementován. Tato podkapitola čerpá z [10].




Obrázek 4.3: Blokové schéma časovače

Modul nastavuje aktivní hodnotu výstupního přerušovacího signálu *p_irq* po uplynutí uživatelsky zadaného počtu hodinových cyklů. Všechny vstupy a výstupy je možné vidět na blokovém schématu časovače na obrázku 4.3. Časovač je řízen a nastavován skrze tyto konfigurační registry:

- **cnt_reg** - uchovává hodnotu čítače, jenž je inkrementován v každém hodinovém cyklu.
- **cmp_reg** - uchovává hodnotu, která je porovnávána s aktuální hodnotou čítače uloženou v registru *cnt_reg* a při jejich shodě dochází k nastavení přerušovacího signálu v následujícím hodinovém cyklu.
- **ctrl_reg** - uchovává mód časovače.

Podle nastaveného módu čítače může být přerušení vystaveno jednou nebo periodicky. Módy časovače jsou následující:

- **DISABLED** - časovač je vypnut, nedochází k navyšování hodnoty čítače uchovávané v konfiguračním registru *cnt_reg*
- **AUTO_RESTART** - při dosažení rovnosti hodnot v konfiguračních registrech *cnt_reg* a *cnt_cmp* dojde v následujícím hodinovém cyklu k nastavení aktivní hodnoty výstupního přerušovacího signálu *p_irq*, k vynulování konfiguračního registru *cnt_reg* a je započato nové čítání hodinových cyklů.

- **ONE_SHOT** - při dosažení rovnosti hodnot v konfiguračních registrech *cnt_reg* a *cnt_cmp* dojde v následujícím hodinovém cyklu k nastavení aktivní hodnoty výstupního přerušovacího signálu *p_irq* k vynulování konfiguračního registru *cnt_reg* a je nastaven mód DISABLED časovače, dojde tedy k jeho vypnutí.
- **CONTINUOUS** - při dosažení rovnosti hodnot v konfiguračních registrech *cnt_reg* a *cnt_cmp* dojde v následujícím hodinovém cyklu k nastavení aktivní hodnoty výstupního přerušovacího signálu *p_irq*. V tomto módu nedochází k vynulování konfiguračního registru *cnt_reg*, časovač čítá dál až do přetečení, čímž dojde k vynulování konfiguračního registru *cnt_reg*.

Kromě generování přerušení v určených hodinových cyklech, lze časovač využít i jako jednoduchý čítač hodinových cyklů. K tomuto slouží registr *cycle_cnt* jehož hodnota je inkrementována po celou dobu běhu časovače, mimo doby trvání aktivního resetu, který je rovněž jediným možným způsobem jak lze tento registr vynulovat.

### 4.3.2 FPU

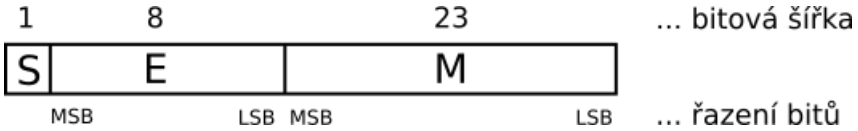
FPU jednotka vytvořená jako modul procesoru Berkelium je implementována s jednoduchou přesností (angl. *single-precision*) a kompatibilní se standardem IEEE-754. Standard byl vytvořen v 80. letech, aby vyřešil problémy spojené s nestandardizovanou reprezentaci čísel s pohyblivou řádovou čárkou (angl. *Floating point number*, FP). Jednoduchá přesnost značí, že tento modul pracuje s 32 bitovou reprezentací operandů. Tato podkapitola čerpá z [1] a [9].

Modul umožňuje následující operace:

- **Aritmetické** - sčítání, odčítání, násobení a dělení
- **Porovnávání** - rovná se, menší než, větší než, menši rovno než, větší rovno než, nerovná se
- **Převodní** - čísla typu float na čísla typu integer, integer na float (float to int, int to float)

#### Reprezentace FP čísel

Standard umožňuje implementaci FPU jednotky s jednoduchou přesností (angl. *single-precision*), dvojitou přesností (angl. *double-precision*) a vícenásobnou přesností, kdy v případě implementace více přesností v FPU jednotce je standardem vyžadována implementace vnitřního přetypování mezi těmito formáty.




Obrázek 4.4: Reprezentace čísla s pohyblivou řádovou čárkou [1]

FP čísla s jednoduchou přesností jsou rozdělena podle standardu IEEE-754 do tří částí odpovídajícím číslu zapsanému v kanonické binární vědecké formě. Rozdělení 32 bitové reprezentace FP čísla je graficky znázorněno na obrázku 4.4, kde jsou využity zkratky MSB pro nejvýznamnější bit z angl. *Most Significant Bit* a LSB pro nejméně významný bit z angl. *Least Significant Bit*. Popis jednotlivých částí je následující:

- Znaménkový bit (angl. *Sign bit*, S) - tento bit je umístěn na pozici 31 (nejvýznamnější bit, angl. *Most Significant Bit*, MSB). Pokud je nastaven na 1, je reprezentované číslo záporné, jinak kladné.
- Exponent (E) - exponent je umístěn na bitech 30 až 23. Exponent normalizovaných čísel je posunut o tzv. bias (E - 127). V případě denormalizovaných čísel, tedy čísel jejichž exponent E je roven 0, je tento bias roven -126.
- Mantisa (M) - mantisa je umístěna na bitech 22 až 0. Mantisa reprezentuje část čísla za řádovou čárkou. S ohledem na kanonickou reprezentaci čísel je k mantise přidán tzv. skrytý bit (angl. *hidden bit*), který je nastaven pro normalizovaná čísla na 1 a pro denormalizovaná čísla na 0.

Tento způsob reprezentace umožňuje rozdělit FP čísla do pěti kategorií uvedených v tabulce 4.3 (X v tabulce značí, že na hodnotě tohoto prvku nezáleží), kde jsou kategorie uvedené přehledně v tomto pořadí:

- **Nula** - z důvodu znaménkového bitu je nula reprezentována dvěma způsoby, tedy +0 a -0.
- **Nekonečno** - kategorie zahrnující $+\infty$ a $-\infty$. Nekonečno se objeví například při dělení nulou.
- **NaN** (Not a Number) - NaN se obvykle objeví v případě, že výsledek operace je nedefinovatelný jako například 0.0 / 0.0. U této kategorie FP čísel nezáleží na znaménku (viz třetí řádek tabulky 4.3). Rozlišuje se více druhů NaN, ale v principu jsou rozlišovány pouze 2 typy, které se rozlišují pomocí MSB mantisy:
  - **sNaN** (signaling NaN, MSB = 0) - produkuje signál obvykle ve formě výjimky od FPU jednotky během výpočtu.
  - **qNaN** (quiet NaN, MSB = 1) - výsledkem je qNaN v případě, že proběhl výpočet v pořádku a NaN je indikován až ve výsledku.

  V implementovaném FPU modulu je určeno, že v případě aritmetické operace nad 2 NaNy je na výstup přenesen NaN ze vstupu A, protože standard neurčuje, který z vstupních NaNů má být převeden na výstup. Pokud je na vstupu A sNaN, je na výstup přenesen qNaN a je signalizován příznak invalid (příznaky viz dále). V případě operace nad sNaN, je výsledkem vždy qNaN a je nastaven příznak invalid. V případě operace nad qNaN není příznak invalid nastaven. qNaN je rovněž výsledkem neplatných aritmetických operací, pro tyto případy je definovaná konstanta symbolizující qNaN - 0xFFC00000.
- **Denormalizovaná čísla** - nenulová FP čísla jejichž exponent odpovídá rezervované hodnotě, která je obvykle rovna možnému minimu (v případě tohoto modulu jde o hodnotu 0).
- **Normalizovaná čísla** - čísla jejichž exponent E je v rozmezí hodnot $0 < E < 255$ a jejich mantisa je nenulová.

Tabulka 4.3: Kategorie čísel podle IEEE-754

| **Znaménkový bit (S, 1 bit)** | **Exponent (E, 8 bitů)** | **Mantisa (M, 23 bitů)** | **Význam** |
|---|---|---|---|
| S | 0 | 0 | S 0 |
| S | 255 | 0 | S $\infty$ |
| X | 255 | Různé od 0 | NaN |
| S | 0 < E <255 | MM....MM (nenulové) | $(-1)^S * 2^{E-127} * (0b1.MM....MM)$ (normalizované číslo) |
| S | 0 | MM....MM (nenulové) | $(-1)^S * 2^{-126} * (0b1.MM....MM)$ (denormalizované číslo) |

### Zaokrouhlování

V implementovaném FPU modulu jsou realizovány všechny typy zaokrouhlování dle standardu IEEE-754. Zaokrouhlování považuje čísla za nekonečně přesné a pokud je to nutné modifikuje je tak, aby je bylo možné reprezentovat cílovým formátem a přitom nastaví příznak inexact. Každá operace by měla vyprodukovat nekonečně přesný mezivýsledek a až poté by mělo dojít k zaokrouhlení podle zvoleného typu zaokrouhlování. Standard IEEE-754 rozlišuje tyto typy zaokrouhlování:

- **Round to nearest** - od implementace standardu IEEE-754 je vyžadován tento typ zaokrouhlování jako výchozí. V případě mezní hodnoty (přesně 0.5), je výsledkem hodnota s nejméně významným bitem (angl. *Least Significant Bit*, LSB) rovným nule. Výsledkem je tedy vždy sudé číslo, kdy v případě vyšší hodnoty dochází k zaokrouhlení směrem nahoru, v případě nižší hodnoty dochází k zaokrouhlení směrem dolů.
- **Round toward $+\infty$** - výsledek je vždy co nejblíže nekonečně přesné hodnotě, ale nikdy není menší. Na zaokrouhlení má vliv znaménko výsledku, kdy kladný výsledek se v případě nepřesnosti zaokrouhluje vždy směrem nahoru a záporný výsledek se zaokrouhluje vždy směrem dolů. Výsledkem není nikdy -$\infty$ a to ani v případě přetečení.
- **Round toward -$\infty$** - výsledek je vždy co nejblíže nekonečně přesné hodnotě, ale nikdy není větší. Na zaokrouhlení má vliv znaménko výsledku, kdy kladný výsledek se v případě nepřesnosti zaokrouhluje vždy směrem dolů a záporný výsledek se zaokrouhluje vždy směrem nahoru. Výsledkem není nikdy $+\infty$ a to ani v případě přetečení.
- **Round toward 0** - pro kladná čísla funguje jako zaokrouhlení k $+\infty$, pro záporná čísla funguje jako zaokrouhlení k -$\infty$. Výsledkem není nikdy +-$\infty$ a to ani v případě přetečení.

### Příznaky

Standard IEEE-754 vyžaduje signalizaci celkem pěti výjimek skrze nastavení příznaků. Tyto příznaky jsou uloženy v registru a mohou být nulovány pouze na vyžádání uživatele. Uživatel by měl mít možnost testovat a měnit příznaky jednotlivě. Registr obsahující příznaky rovněž uchovává informaci o zvoleném typu zaokrouhlování. Standardem vyžadovanými příznaky jsou:

- **Invalid** - nastaven v případě operace nad sNaN nebo v případě neplatných aritmetických operací.
- **Div_zero** - nastaven v případě, že dělitel je 0 a dělenec je konečné nenulové číslo. V případě 0.0 / 0.0 je výsledkem nekonečno bez nastavení příznaků inexact a overflow.
- **Overflow** - nastaven v případě, že výsledek je větší než největší zobrazitelné číslo. Výsledkem je nekonečno nebo +- největší zobrazitelné číslo podle typu zaokrouhlení. Pokud je jeden z operandů nekonečno, příznak se nenastavuje. ($\infty$ + číslo je přesně $\infty$ - příznak overflow není nastaven).
- **Underflow** - nastaven v případě, že výsledek je denormalizované číslo a současně došlo k nastavení příznaku inexact. Příznak underflow je v této implementaci vyhodnocen vždy po zaokrouhlení. V případě, že je výsledek (po zaokrouhlení) nejmenší normalizované číslo $(2^{-126}, 2^{126})$ není příznak underflow nastaven.
- **Inexact** - nastaven v případě, že výsledek není stejný jako výsledek počítaný s nekonečnou přesností, výsledkem je zaokrouhlené číslo. Operace nad nekonečnem jsou přesné ($\infty$ + číslo je přesně $\infty$ - příznak inexact není nastaven).

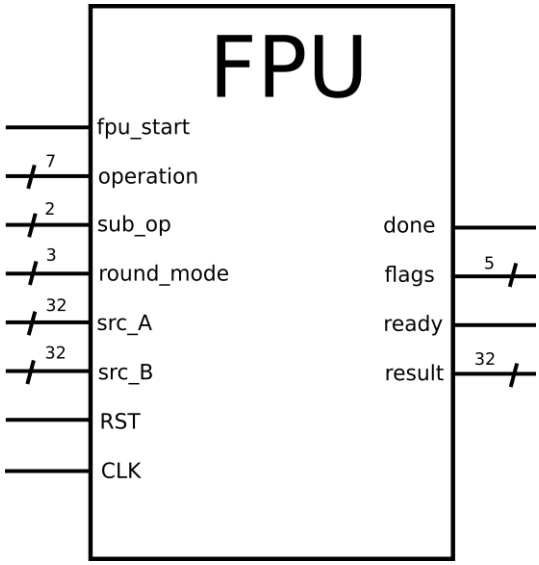
V tabulce 4.4 je zobrazeno, které příznaky mohou být nastaveny jednotlivými operacemi.

Tabulka 4.4: Operace a nastavované příznaky

| **Operace** | **Příznaky** |
|---|---|
| +, –, * | všechny kromě div_zero |
| / | všechny |
| int to float | inexact |
| float to int | invalid, inexact |
| porovnání | invalid |
| MinMax | invalid |
| Class | žádné |

### Operace FPU

Pro provedení výpočtu v modulu FPU je nejdříve potřeba přivést na jeho vstupy operandy, požadovanou operaci nad nimi a způsob jakým má být výsledek výpočtu zaokrouhlen. Všechny vstupy a výstupy je možné vidět na blokovém schématu FPU na obrázku 4.5. Operace se nastavují pomocí 8 bitového vstupu *operation* a zároveň pro některé z operací je ještě nutné nastavit 2 bitový vstup *sub_op*, který dále specifikuje tyto operace mezi něž patří například porovnávání, kdy je pomocí signálu *sub_op* nastaven typ porovnání ( rovnost, menší než, menší rovno). Nastavením signálu *fpu_start*, potvrzujícím platnost údajů přivedených na vstupy FPU modulu, dojde k spuštění výpočtu v případě, že modul signalizuje připravenost na provedení dalšího výpočtu pomocí výstupního signálu *ready*. V případě, že je signál *fpu_start* nastaven v průběhu probíhajícího výpočtu, je tento požadavek ignorován.




Obrázek 4.5: Blokové schéma FPU

Před zahájením samotné operace nad operandy jsou vstupní údaje uloženy do registrů FPU modulu. Po dokončení výpočtu FPU modul přivede na výstup odpovídající data a potvrdí jejich platnost nastavením výstupního signálu *done*. Operace probíhají v těchto krocích (nebo alespoň v několika z nich):

- **pre-normalizace** - jsou prováděny úpravy vstupních operandů potřebné pro správný výpočet výsledku požadované operace nad nimi.
- **samotná operace** - je prováděn algoritmus odpovídající zvolené operaci nad vhodně upravenými operandy.
- **post-normalizace** - jsou prováděny úpravy výsledku do podoby v jaké je potřeba na výstupu.
- **zaokrouhlení** - v určitých případech je třeba provádět zaokrouhlení podle zvoleného typu zaokrouhlování.
- **případná normalizace po zaokrouhlení** - je provedena, pokud během zaokrouhlování dojde ke změně výsledku do podoby, v jaké nemůže být přiveden na výstup modulu a je tedy potřebná jeho opětovná normalizace.
- **výjimky a nastavení příznaků** - jsou nastaveny odpovídající příznaky.

Každá z operací má jinou pevnou latenci (dobu od začátku výpočtu, až po dokončení a přivedení platných údajů na výstup modulu) odpovídající délce samotné operace a latencím vstupních a výstupních registrů. Latence výpočtu je stejná i v speciálních případech, kdy je možné přivést výsledek na výstup prakticky okamžitě (např. jeden ze vstupních operandů je NaN).

# Kapitola 5

# Zadání úlohy

Během vývoje hardwarových systémů je třeba provádět opakovaně kontrolu jejich funkčnosti vzhledem k zadané specifikaci, neboť i malá změna v návrhu může vnést nové chyby do vyvíjeného hardwarového systému. Tyto kontroly jsou prováděny opakovaně a je tedy důležité, aby jejich časová náročnost byla co nejmenší a neprodloužila tak výrazně dobu vývoje systému. Pro kontrolu vyvíjených hardwarových systémů se v dnešní době nejvíce využívá funkční verifikace.

Doba trvání verifikace roste s úrovni abstrakce hardwarového systému, na které je prováděna. Hlavními problémy funkční verifikace jsou doba trvání simulace a pseudonáhodné generování stimulů, kterých využívá. Simulace přirozeně paralelních hardwarových systémů se prodlužuje s jejich rostoucí komplexností a může trvat i několik dní. Pseudonáhodné generování stimulů, je žádoucí z pohledu verifikace, neboť může odhalit chování systému na nějž nebylo myšleno při psaní jeho specifikace. Tento přístup ke generování stimulů však zanáší do verifikace další nežádoucí zpomalení, neboť dochází k vysoké redundanci, tedy k opakovanému generování některých stimulů. Některé vlastnosti jsou tedy ověřeny vícekrát, zatímco paradoxně k verifikaci některých okrajových případů nedochází vůbec, nebo za velmi dlouhou, předem neurčitelnou dobu. Vysoká míra redundance znemožňuje využití pseudonáhodného generování stimulů při verifikaci na systémové úrovni. Neustále narůstající komplexnost hardwarových systémů čím dal více ztěžuje provádění verifikace efektivně, neboť narůstá nejen časová náročnost jejího trvání, ale i samotné přípravy odpovídající verifikace. Toto je hlavním důvodem vývoje stále dokonalejších technik pro její zefektivnění.

Cílem této práce je prozkoumat možnosti portovatelných stimulů, jako jedné z nově vytvářených technik (podpořených standardem). Jedním z dílčích cílů je vyhodnocení míry snížení redundance při generování pseudonáhodných stimulů, kdy tato technika využívá řízeného generování stimulů pomocí sady definovaných pravidel. Dalším dílčím cílem je pak vyhodnocení portovatelnosti při využití této techniky napříč úrovněmi verifikovaného systému. Pro zkoumání možností portovatelných stimulů byl vybrán nástroj Questa InFact od společnosti Mentor, jenž je jednou z předních společností v této oblasti a od této společnosti byl rovněž vybrán simulační nástroj QuestaSim pro měření a zobrazení pokrytí. Pro dosažení požadovaných cílů je třeba provést hned několik kroků. Nejdříve je potřeba nastudovat verifikační prostředí vytvářená v jazyce SystemVerilog s využitím metodiky UVM, verzi jazyka pro vytváření portovatelných stimulů v nástroji Questa InFact a možnosti samotného nástroje. Dalším krokem je pak návrh a implementace verifikačních scénářů portovatelných stimulů pro vybrané hardwarové obvody, přičemž v této práci budou využity moduly procesoru Berkelium společnosti Codasip. Po odladění budou tyto verifikační scénáře přeneseny a následně využity při společné hardwarové a softwarové verifikaci na systémové úrovni, kde budou tyto moduly připojeny k procesoru Berkelium 5.

# Kapitola 6

# Návrh a implementace verifikačních scénářů

Tato kapitola popisuje návrh a implementaci verifikačních scénářů portovatelných stimulů pro vybrané moduly procesoru. Dále je zde popsán způsob, jakým bylo provedeno portování implementovaných verifikačních scénářů pro verifikaci na systémové úrovni. Verifikační prostředí pro jednotlivé vybrané moduly jsou psána v jazyce SystemVerilog za použití metodiky UVM a tomu i odpovídají soubory generované nástrojem Questa InFact pro vytvářené verifikační scénáře.

## 6.1 Úpravy verifikačního prostředí

Pro tuto práci jsou poskytnuta již hotová verifikační prostředí, které je třeba upravit, aby bylo možné připojit verifikační scénáře portovatelných stimulů implementované pro vybrané moduly. V nástroji Questa InFact jsou verifikační scénáře generovány samostatnou testovací komponentou připojitelnou k verifikačnímu prostředí (viz kapitolu 3.2.5). Tento krok je proveden ihned po vytvoření základní kostry verifikačního scénáře, aby bylo možné provádět ladění během jeho implementace.

Hierarchie souborů projektu (modulů) je vyobrazena na obrázku 6.2, kde složka *rtl* uchovává DUT v jednom z HDL jazyků, složka *uvm_user* obsahuje veškeré soubory tvořící verifikační prostředí a složka se sufixem *infact* obsahuje testovací komponenty generující implementované verifikační scénáře nástrojem Questa InFact. U poskytnutých verifikačních prostředí musely být upraveny nebo přidány tyto soubory:

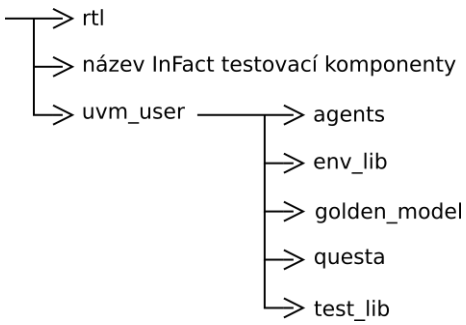
- uvm_user/test_lib/**test.svh** - do tohoto souboru byla přidána nová testovací třída obsahující spuštění testovací komponenty generující implementované verifikační scénáře stejným způsobem, jakým jsou spouštěny verifikační sekvence při použití metodiky UVM.
- uvm_user/questa/**start_common.tcl** - jeden ze spouštěcích skriptů, ve kterém probíhá výběr spouštěného testu. Je třeba vybrat nově vytvořený test, který v sobě obsahuje spuštění testovací komponenty.
- uvm_user/questa/**start_gui.tcl** - do tohoto spouštěcího skriptu, bylo třeba přidat parametry pro spuštění simulátoru, které mimo jiné dovolují algoritmu nástroje Questa InFact řízení verifikačních scénářů generovaných testovací komponentou. Tyto

přidané parametry jsou uvozeny `+infact=`. Na obrázku 6.1 je vyobrazena tato úprava pro modul časovače. Kromě automaticky vygenerovaného inicializačního souboru nástrojem Questa InFact, je také přidán vlastní inicializační soubor (viz dále).

```
+infact=../timer_infact/timer_infact.ini +infact=./infact.ini"
```

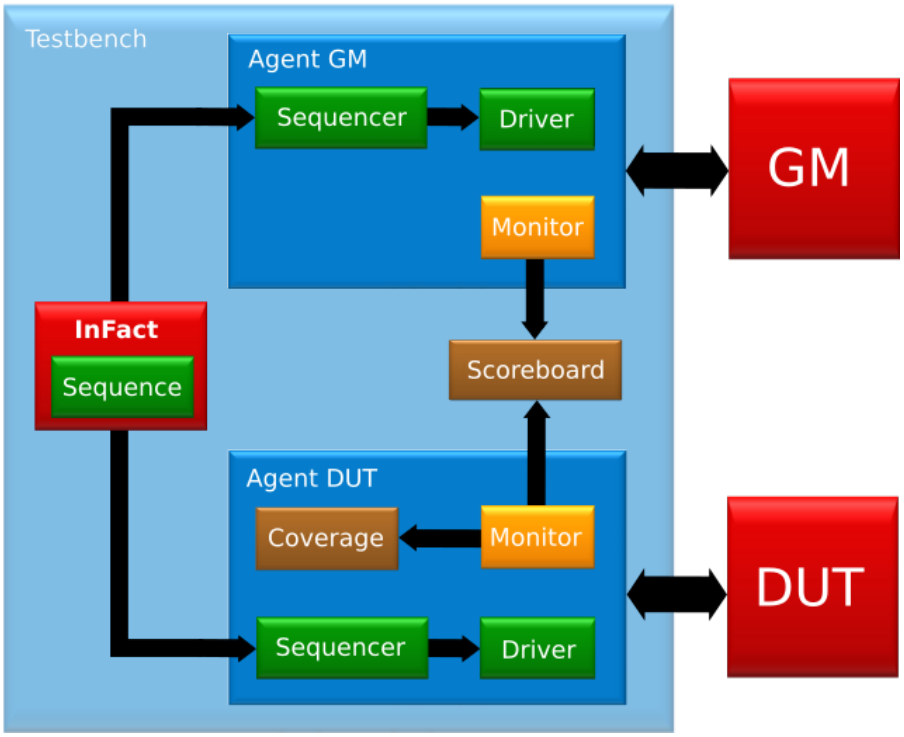
Obrázek 6.1: Parametry přidané pro spuštění simulátoru

- uvm_user/agents/**sv_agent_pkg.sv** - do tohoto souboru musel být přidán include vygenerovaného souboru v jazyce SystemVerilog pro testovací komponentu.
- uvm_user/**uvm_files.f** - zde musela být přidána cesta ke složce obsahující vytvořenou testovací komponentu.
- uvm_user/**infact.ini** - vlastní inicializační soubor vytvořený pro potřebu explicitně definovat *seed* pro generátor hodnot verifikačních scénářů. Verifikační prostředí těchto modulů implementuje spouštění sekvencí v testovací třídě pomocí konstrukce *fork*, kdy jsou vytvořeny dva procesy běžící paralelně. V těchto procesech jsou vytvářeny stejné sekvence, ale jeden proces posílá transakce vytvářené těmito sekvencemi na vstupy GM a druhý na vstupy DUT. Tímto způsobem je implementována kontrola za běhu (angl. *on-the-fly-checking*). Nástroj Questa InFact doposud nepodporuje globální definici jednoho *seedu* a je nutné tedy definovat tento *seed* explicitně pro každé spuštění testovací komponenty generující implementované verifikační scénáře, i když oba procesy spouští tutéž testovací komponentu.
- uvm_user/**custom_seeds** - přidaný soubor obsahující explicitní definice *seedu* pro každé spuštění testovacích komponent v implementované testovací třídě, která je obsahuje.




Obrázek 6.2: Hierarchie souborů projektu

Jak vypadá zapojení testovací komponenty generující implementované verifikační scénáře do verifikačního prostředí pro ovládání generování stimulů je ukázáno na obrázku 6.3.




Obrázek 6.3: Zapojení testovací komponenty do verifikačního prostředí

## 6.2 Verifikační scénáře pro modul časovače

Implementace verifikačních scénářů pro modul časovače je pro přehlednost rozdělena do více souborů, ze kterých je posléze vygenerována výsledná testovací komponenta. Kromě segmentového souboru obsahujícího strukturu, odpovídá toto rozdělení rovněž rozdělení problému definování verifikačních scénářů pro odlišné vlastnosti časovače, jenž mají být verifikovány. Hlavní metrikou pro návrh a implementaci verifikačních scénářů byla implementace funkčního pokrytí ve verifikačním prostředí časovače.

### 6.2.1 Struktura

Prvním krokem při návrhu a implementaci verifikačních scénářů je definice meta akcí představujících signály DUT. V případě připojování testovací komponenty k verifikačnímu prostředí časovače, které je implementováno v HVL jazyce SystemVerilog za pomoci metodiky UVM, jsou tyto meta akce uvnitř struktury, jenž je napojena na třídu `timer_transaction` představující transakci ve verifikačním prostředí. Struktura samotná je definována v segmentovém souboru `timer_struct.rseg`, který mimo jiné obsahuje také definice používaných konstant a definici rozhraní nazvanou `do_item` sloužící k propojení pro přenos transakcí generovaných testovací komponentou do verifikačního prostředí časovače.

Kromě meta akcí jsou ve struktuře definovány také pro některé z nich biny implicitního typu, které určují algoritmu nástroje, jaké hodnoty a v jakých rozsazích mohou být generovány. Například v případě meta akce `RST`, která má definován rozsah hodnot pomocí konstrukce množiny by byla definice binů zbytečná, protože může nabývat pouze dvou hodnot. Naopak v případě meta akce `if_dbus_DO`, která má rozsah definován pomocí bitové

šířky a může nabývat hodnot 0 až $2^{32}-1$, by při zaměření této meta akce ve strategii pokrytí bylo vygenerováno mnoho verifikačních scénářů, lišících se pouze hodnotou této meta akce, kdyby nebyl definovaný obsah rozdělen do 64 binů. V tomto případě je rozdělení provedeno pomocí operátoru /.

Je třeba dát pozor na to, aby biny nevylučovaly hodnoty, které jsou generovány pomocí constraintů nebo pomocí rozsahů definovaných přímo v sekvenci pravidel. To lze vidět například u meta akce `if_dbus_A0`, kde z hlediska funkcionality je důležitých jen pár hodnot odpovídajících adresám registrů v časovači, ale zároveň je mimo strukturu definován statický constraint vyžadující právě hodnoty lišící se od adres registrů. Je tedy potřeba při návrhu verifikačních scénářů myslet i na případy, kdy tento signál bude nabývat nežádoucích či zbytečných hodnot. Pro tuto meta akci je proto definován i bin s operátorem *, jenž vytvoří jediný bin pro všechny, ostatními biny explicitně nepokryté, hodnoty. Uvedené meta akce jsou spolu s definovanými biny ukázány na obrázku 6.4, kde `logic` je název množiny [0,1] vytvořené pro zjednodušení zápisu.

```
meta_action RST [logic];
meta_action if_dbus_A0 [unsigned ADDR_WIDTH-1:0];
bins if_dbus_A0
    [TIMER_CNT]
    [TIMER_COMP]
    [TIMER_CR]
    [TIMER_CYCLE_L]
    [TIMER_CYCLE_H]
    [*];
                    •
                    •
                    •
                    •
meta_action if_dbus_D0 [unsigned DATA_WIDTH-1:0];
bins if_dbus_D0/64;
```

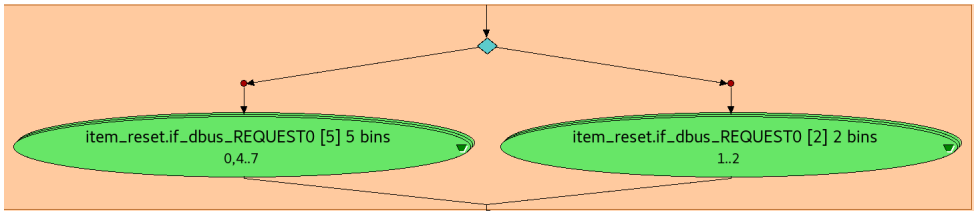
Obrázek 6.4: Různé definice binů meta akcí ve struktuře

Ve struktuře jsou dále definovány také konstrukce `req` a `count` typu symbol, které zastřešují větvení obohacené o pravděpodobnostní tagy pro generování různých hodnot meta akcí. Konstrukce `req` má dvě větve a v obou větvích je generována hodnota pro meta akci `if_dbus_REQUEST0`, zároveň je ale v každé větvi výčtem určeno jakých hodnot může při jejím průchodu nabývat s tím, že vyšší pravděpodobnost je dána požadavkům na čtení a zápis. Konstrukce `count` má tři větvě a ve všech je generována hodnota pro meta akci `if_dbus_D0`. V tomto případě jsou hodnoty určené ve dvou větvích pomocí rozsahů a v poslední je určena jedna konkrétní hodnota. Nejvyšší pravděpodobnost je pak dána větvi, která pro meta akci generuje hodnoty z rozsahu 1 až 255. Definice konstrukce `req` spolu s pravidly, které obsahuje je ukázána na obrázku 6.5 a dále pak na obrázku 6.6 je ukázána její grafová reprezentace.

```
symbol req =
    tag prob_1 if_dbus_REQUEST0[CP_CMD_NONE, CP_CMD_INVALIDATE, CP_CMD_INVALIDATE_ALL, CP_CMD_FLUSH, CP_CMD_FLUSH_ALL]|
    tag prob_15 if_dbus_REQUEST0[CP_CMD_READ, CP_CMD_WRITE];
```

Obrázek 6.5: Kód konstrukce `req` implementované uvnitř struktury

Struktura také obsahuje definici několika constraintů. Jedním z nich je statický constraint určující, že meta akce RST bude vždy nabývat neaktivní hodnoty resetu. Tento constraint je zaveden, protože není žádoucí, aby vytvářené transakce náhodně aktivovaly reset časovače. Díky tomu, že je definován uvnitř struktury, je automaticky uplatněn ve všech jejích vytvořených instancích. Další constrainty obsažené ve struktuře jsou dynamického typu a mají za cíl nastavit některé z meta akcí na správné či užitečné hodnoty (na



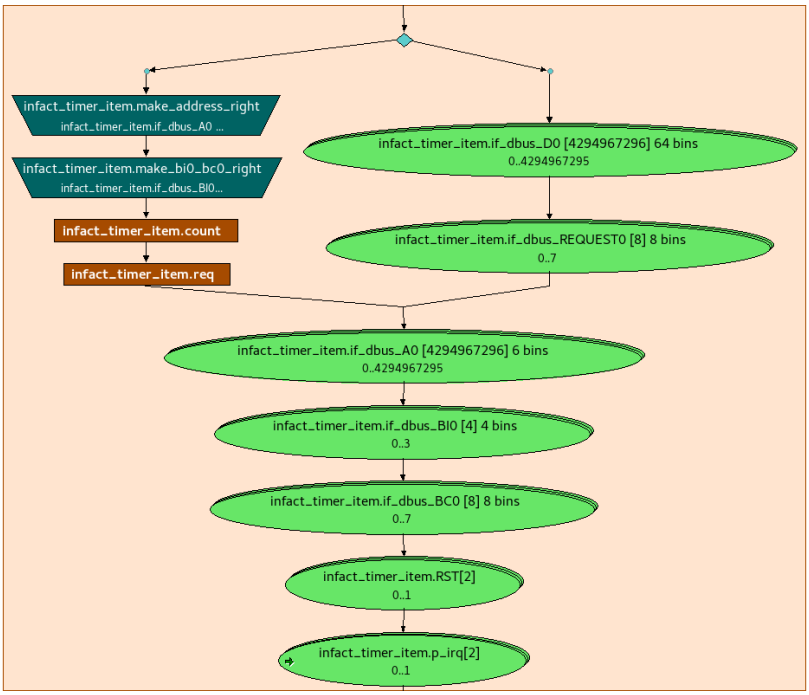

Obrázek 6.6: Grafová reprezentace konstrukce `req` implementované uvnitř struktury

obrázku 6.7a je lze vidět na řádcích 100 a 101). Poslední část struktury je definice sekvence pravidel pro generování hodnot jednotlivých meta akcí. Je důležité rozlišovat meta akce, které představují jednotlivé stimuly posílané na signály časovače a strukturu, která představuje celou transakci obalující tyto stimuly.

Definice sekvence pravidel pro generování meta akcí uvnitř struktury je nepovinná, avšak dává možnost ovlivňovat vytvářené verifikační scénáře jak na úrovni transakcí, tak na úrovni samotných signálů. Zde je využita tato možnost pro vytvoření zcela náhodného generování hodnot pro signály časovače a zároveň pro vytvoření náhodných, ale správných a užitečných hodnot pro signály časovače. Toho je dosaženo větvením, kdy jedna z větví obsahuje dynamické constrainty pro zajištění odpovídajících hodnot při průchodu touto větví a také obsahuje výše popsané konstrukce `req` a `count`. Druhá větev naopak není nijak ovlivňována a může generovat libovolné hodnoty, třeba i správné. Výsledná podoba definice sekvence pravidel uvnitř struktury je na obrázku 6.7a a na obrázku 6.7b je ukázána její grafová reprezentace s nerozbalenými konstrukcemi `req` a `count`.

```
 98 timer_transaction =
 99 (
100     make_address_right
101     make_bi0_bc0_right
102     count
103     req
104 )
105 |
106 (
107     if_dbus_D0
108     if_dbus_REQUEST0
109 )
110 if_dbus_A0
111 if_dbus_BI0
112 if_dbus_BC0
113 RST
114 p_irq
115 ;
```

(a) Definice kódu sekvence pravidel struktury




(b) Grafová reprezentace sekvence struktury

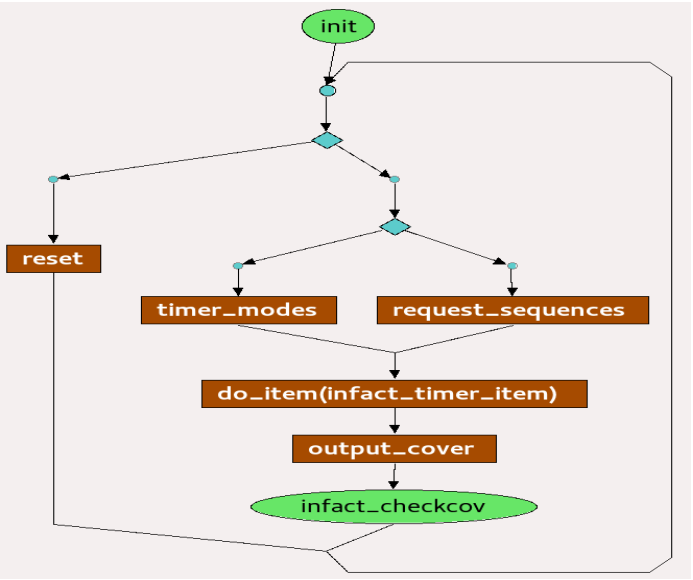
Obrázek 6.7: Výsledná struktura

### 6.2.2 Verifikační scénáře

Všechny navržené a následně implementované verifikační scénáře jsou spojeny v hlavní sekvenci pravidel tvořící výsledný graf. Podoba této sekvence v pravidlech je vyobrazena na obrázku 6.8a a její grafová reprezentace je pak na obrázku 6.8b. Dále jsou postupně popsány všechny podgrafy, které reprezentují různé verifikační scénáře.

```
infact_timer_seq = init repeat{
    (
        reset
    )
    |
    (
        (timer_modes)|
        (request_sequences)
        do_item(infact_timer_item)
        output_cover
        infact_checkcov
    )
};
```

(a) Definice kódu hlavní sekvence pravidel




(b) Podoba výsledného grafu

Obrázek 6.8: Výsledný graf

Podgraf, jenž je tvořen pravidly uvnitř konstrukce `symbol` nesoucí název `timer_modes`, je zodpovědný za generování scénářů verifikujících jednotlivé nastavitelné módy časovače. Celá implementace tohoto podgrafu je v souboru *timer_modes_rules.rseg*.

Pro nastavování a spouštění jednotlivých módů bylo potřeba vytvořit pět instancí struktury. Pomocí nich nejdříve postupně dojde k zastavení počítání časovače nastavením módu *DISABLED*, vynulování hodnoty čítače v registru *TIMER_CNT*, nastavení hodnoty, do které se má počítat v registru *TIMER_COMP* a je spuštěno čítání nastavením aktivního módu čítače v registru *TIMER_CR*. Kromě nastavení módu časovače, je vše provedeno s využitím statického constraintu. Hodnota, do které má být počítáno, je zvolena náhodně algoritmem nástroje z rozsahu 5 až 10. Dalším krokem se stalo navržení způsobu, jakým budou jednotlivé módy časovače vybírány. Díky možnostem jazyka nástroje bylo těchto způsobů navrženo hned několik. Nakonec byl vybrán návrh využívající dynamických constraintů. Bylo vytvořeno větvení a do každé větve byl umístěn dynamický constraint nastavující jiný mód časovače. Navíc bylo potřeba vytvořit pro každou větev akci, která funguje jako značka pro algoritmus nástroje a to tak, že je zahrnuta do strategie pokrytí. Tím, že jsou tyto akce součástí strategie pokrytí, algoritmus nástroje navštíví každou z nich a je tak zajištěna tvorba verifikačního scénáře pro všechny módy časovače.

Součástí tohoto podgrafu je také smyčka generující transakce čekajíc na vygenerování přerušení časovačem na výstupu `p_irq`. Smyčka je implementována pomocí klíčového slova `repeat` a parametru určujícím počet opakování. V případě této implementace je využito jako parametru klíčové slovo `action_limited`, které udává, že smyčka může být ukončena pouze pomocí metody grafu *stop_loop_expansion_after()*. Uvnitř smyčky je umístěna in-

stance struktury `item_count` zodpovědná za generování transakcí při čekání a rovněž je určena pro zaregistrování čekaného přerušení. Pomocí statického constraintu je pro tuto instanci určeno, že může generovat pro adresový vstup pouze hodnoty vyšší než 20, aby při čekání nedošlo k přepsání některého z registrů a nebyla tak ovlivněna doba čekání na přerušení nebo změněn mód časovače. Dále je pak uvnitř smyčky akce `break_count`, ve které je implementováno ukončení smyčky. Tato implementace je umístěna přímo v souboru *timer_modes_rules.rseg* pomocí konstrukce `attributes`, kde přiřazení řetězce ke klíčovému slovu `action_stmt` znamená, že celý obsah metody napojené na tuto akci jím bude nahrazen (viz obrázek 6.9).

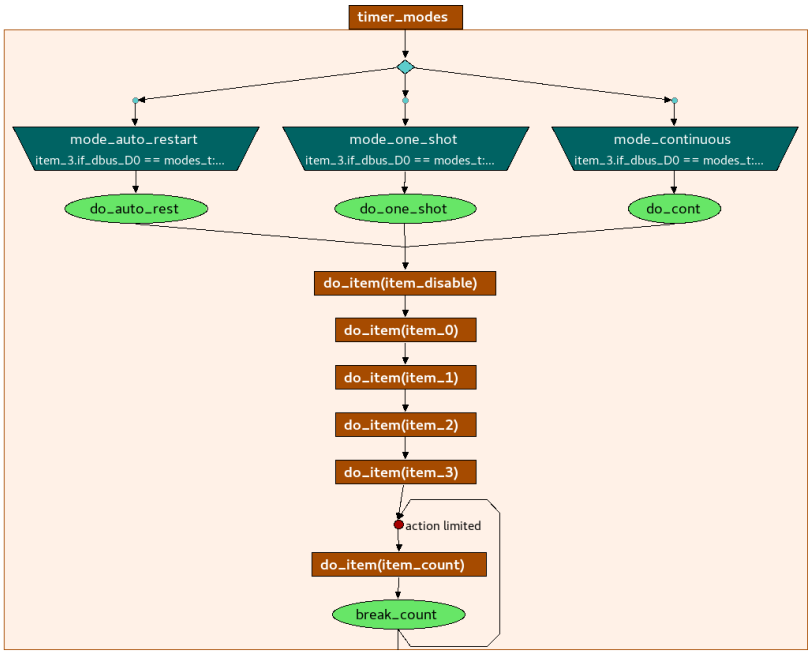
```
12    attributes break_count{
13        action_stmt =
14        "
15        if(m_item_count.p_irq == 1) begin
16            m_te.stop_loop_expansion_after(2,0);
17        end
18        "
19    }
```

Obrázek 6.9: Kód metody akce `break_count`

Jakmile je zaregistrováno přerušení pomocí instance struktury `item_count`, akce `break_count` zavolá metodu grafu *stop_loop_expansion_after()*. Parametry této metody jsou nastaveny tak, aby byla smyčka provedena před svým ukončením ještě dvakrát a byly tak vygenerovány ještě dvě transakce. Generování dvou dalších transakcí je zde pro verifikování správného chování časovače poté co dojde k vygenerování přerušení na výstupu `p_irq` v jeho jednotlivých módech. Výsledná podoba definice sekvence pravidel pro tento podgraf je na obrázku 6.10a a na obrázku 6.10b je pak ukázán samotný podgraf.

```
symbol timer_modes =
        (mode_auto_restart do_auto_rest)
        |
        (mode_one_shot do_one_shot)
        |
        (mode_continuous do_cont)

        do_item(item_disable)
        do_item(item_0)
        do_item(item_1)
        do_item(item_2)
        do_item(item_3)
        repeat action_limited
        {
            do_item(item_count)
            break_count
        }
        ;
```

(a) Definice kódu sekvence pravidel




(b) Grafová reprezentace sekvence pravidel

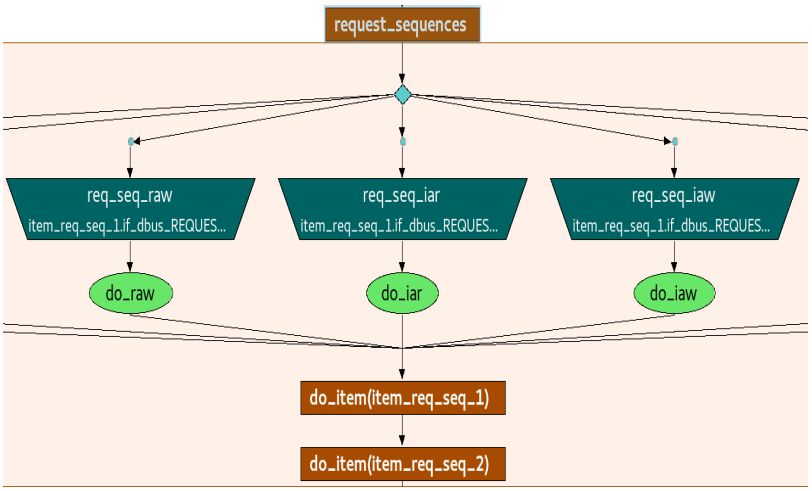
Obrázek 6.10: Verifikace módů časovače

Podgraf, jenž je tvořen pravidly uvnitř konstrukce `symbol` nesoucí název `request_sequences`, je zodpovědný za generování scénářů verifikujících všechny možné kombinace požadovaných operací časovače v po sobě jdoucích transakcích, mezi nimiž je například RAW (Read after Write). Celá implementace tohoto podgrafu je v souboru *timer_request_seq_rules.rseg*.

K definici pravidel těchto scénářů byly vytvořeny dvě instance struktury, které generují odpovídající požadavky na vstupech v po sobě následujících taktech. Pro výběr verifikované sekvence v daném běhu bylo využito dynamických constraintů. Stejně jako při verifikaci jednotlivých módů časovače bylo vytvořeno větvení a do každé větve byl umístěn odpovídající dynamický constraint a akce, která byla zahrnuta do strategie pokrytí, aby bylo zajištěno vykonání každé větve. Byl vyzkoušen i návrh využívající namísto tohoto větvení konstrukce strategie pokrytí *path coverage*, která zahrnovala meta akce `if_dbus_REQUEST0` z obou instancí struktury. Nástroj Questa InFact úspěšně určil správný počet kombinací požadavků operací, ale simulátor QuestaSim, jenž je pro tuto práci používán, nedokázal provést takto definované generování kombinací. Jak vypadá sekvence pravidel pro tyto verifikační scénáře je na obrázku 6.11a a odpovídající grafová reprezentace je pak na obrázku 6.11b.

```
symbol request_sequences =
        (req_seq_rar do_rar)|
        (req_seq_war do_war)|
        (req_seq_raw do_raw)|
        (req_seq_iar do_iar)|
        (req_seq_iaw do_iaw)|
        (req_seq_rai do_rai)|
        (req_seq_wai do_wai)
        do_item(item_req_seq_1)
        do_item(item_req_seq_2)
        ;
```

(a) Definice kódu sekvence pravidel




(b) Grafová reprezentace sekvence pravidel

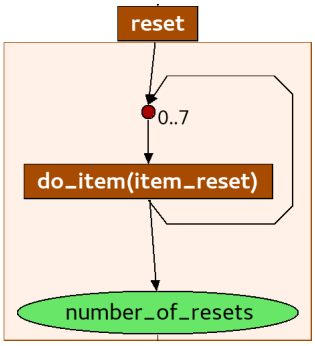
Obrázek 6.11: Verifikace kombinací požadavků na časovač

Podgraf, jenž je tvořen pravidly uvnitř konstrukce `symbol` nesoucí název `output_cover`, byl vytvořen pro generování scénářů, které na výstup časovače `if_dbus_Q0` dodají všechny hodnoty z definovaných rozsahů, jejichž disjunkce dává rozsah definovaný u meta akce `if_dbus_Q0`. Celá implementace tohoto podgrafu je v souboru *timer_output_cover_rules.rseg*.

Sekvence pravidel tohoto podgrafu je tvořena dvěma instancemi struktury. První instance generuje požadavek na zápis na specifikovanou adresu, v případě této implementace jde o adresu odpovídající konfiguračnímu registru v němž se uchovává mód timeru, a druhá následně provádí čtení z této adresy. Pro správné nastavení těchto instancí je použito statického constraintu. Dosažení pokrytí všech nadefinovaných hodnot je provedeno zahrnutím meta akce *if_dbus_D0* v pořadí druhé instance do strategie pokrytí, tedy instance, která generuje požadavek na čtení. Strategie pokrytí je definována na meta akci generující vstupy časovače, protože jejím úkolem je směrovat generování hodnot meta akcí nástrojem Questa InFact, nejedná se o pouhou definici metriky.

Podgraf, jenž je tvořen pravidly uvnitř konstrukce `symbol` nesoucí název `reset`, byl vytvořen pro náhodné vkládání resetovací sekvence v průběhu simulace. Celá implementace tohoto podgrafu je v souboru *timer_reset_rules.rseg*.

Tato sekvence obsahuje pouze jednu instanci struktury uvnitř smyčky. Smyčka je definována kličovým slovem `repeat` následované parametrem `[0..7]`, který ponechává určitou volnost algoritmu nástroje při určení počtu taktů s resetem nastaveným na aktivní hodnotu. V tomto podgrafu je také akce `number_of_resets`, která podobně jako v předchozích případech slouží jako značka pro algoritmus nástroje (je součástí strategie pokrytí), s tím rozdílem, že počet požadovaných zásahů této akce algoritmem nástroje je zvýšen na 3 (výchozí hodnota počtu zásahů je rovna 1). Problém při definici pravidel pro tento podgraf spočíval v nastavení aktivní hodnoty resetu. Ve struktuře jejíž instance jsou používány ke generování transakcí, je statický constraint určující všem jejím instancím nastavení resetu na neaktivní hodnotu. Zde bylo využito toho, že constrainty ovlivňují vybírání hodnot pro meta akce algoritmem nástroje, ale již neomezují jejich případnou změnu. Byl přidán atribut pro instanci struktury, který nahrazuje hodnotu generovanou algoritmem nástroje pro meta akci `RST` neaktivní hodnotou resetu. Grafová reprezentace tohoto podgrafu je na obrázku 6.12.




Obrázek 6.12: Grafová reprezentace resetu

V hlavním souboru *infact_timer_seq.rules* obsahujícím definici sekvence představující celý graf jsou rovněž definovány akce `init`, `infact_checkcov` a instance struktury `infact_timer_item`. Tyto akce jsou automaticky vygenerovány při tvorbě testovací komponenty. Akce `init` je vygenerována prázdná a umístěna před začatek hlavní smyčky grafu. V této implementaci je využita pro nastavení vah pravděpodobnostních tagů. Pro akci `infact_checkcov` je nástrojem vygenerována metoda, která při průchodu touto akcí kontroluje, zda již bylo dosaženo cílů nastavených pomocí strategie pokrytí a rovněž ukončuje hlavní smyčku grafu při jejich dosažení. Instance `infact_timer_item` byla implementována pro vytvoření základních cílů strategie pokrytí, kde generování hodnot všech definovaných meta akcí této instance není nijak ovlivněno, závisí pouze na parametrech a omezeních definované sekvence struktury. Kromě meta akcí pro výstup přerušení a resetovací vstup, zahrnuje strategie pokrytí všechny meta akce této instance. U většiny těchto meta akcí bylo pro jejich pokrytí dostačující využít konstrukce *pokrytí akce* (angl. *action coverage*), u meta akcí `if_dbus_BI0` a `if_dbus_BC0`, bylo nutné využít konstrukce *pokrytí cesty* (angl. *path coverage*), aby došlo k plnému strukturnímu pokrytí podmínek.

## 6.3 Verifikační scénáře pro modul FPU

Návrh a implementace verifikačních scénářů pro modul FPU probíhal z počátku podobně jako u modulu časovače v rámci jedné testovací komponenty. S narůstajícím počtem verifikačních scénářů však bylo nakonec provedeno rozdělení do více menších testovacích komponent, kdy každá z nich zodpovídá za generování verifikačních scénářů pro pokrytí jiných aspektů modulu FPU během verifikace.

Je třeba si uvědomit, že verifikační scénáře jsou algoritmem nástroje generovány pouze s určitou mírou řízení pomocí definované strategie pokrytí, stále je však využíváno pseudonáhodného generování. Může se tedy stát, že po přidání nových pravidel přestanou verifikační scénáře, generované pomocí dříve definovaných pravidel, pokrývat části, pro které již byly odladěny. Pokud se tak stane, znamená to, že v těchto pravidlech jsou nedostatky kryté pomocí náhodnosti generování. Oprava těchto pravidel může trvat relativně dlouho a je provedena vždy určitou změnou pravidel (změna constraintů, přidání instancí struktury atd.), což může opět odhalit další nedokonalosti vyskytující se jinde. V případě velkého množství pravidel se tento jev stává velkou časovou zátěží, a proto bylo pro jeho omezení zvoleno rozdělení jedné velké testovací komponenty do několika menších.

Hlavní metrikou pro návrh a implementaci verifikačních scénářů pro tento modul bylo strukturní pokrytí generované simulátorem.

### 6.3.1 Struktura

Stejně jako v případě modulu časovače jsou definovány meta akce odpovídající signálům DUT uvnitř struktury, která je napojena na třídu `devel_32f_transaction` ve verifikačním prostředí modulu FPU představující transakci. Struktura je definována v segmentovém souboru `fpu_common.rseg`, který mimo jiné obsahuje také definice používaných konstant a definici rozhraní nazvanou `do_item` sloužící k propojení pro přenos transakcí generovaných testovací komponentou do verifikačního prostředí FPU modulu. Definice této struktury je pro všechny implementované testovací komponenty stejná a je tedy umístěna mimo ně, aby mohla být snadno importována do všech.

Pro několik meta akcí uvnitř struktury jsou implementovány biny implicitního typu, které v případě návrhu a implementace verifikačních scénářů pro FPU modul slouží jako výchozí. Je zde také definován statický constraint, který určuje pro meta akci `RST` neaktivní hodnotu resetu a pro meta akci `fpu_start` hodnotu spouštějící operaci FPU modulu všem instancím této struktury.

Součástí souboru, ve které je struktura definována, je také několik dalších konstrukcí společných všem testovacím komponentám. Jsou zde definovány tři instance struktury z nichž dvě jsou zavedeny pro vkládání resetovacích sekvencí během simulace a poslední je pak zodpovědná za generování transakcí během čekání na dokončení operace, běžící v FPU modulu.

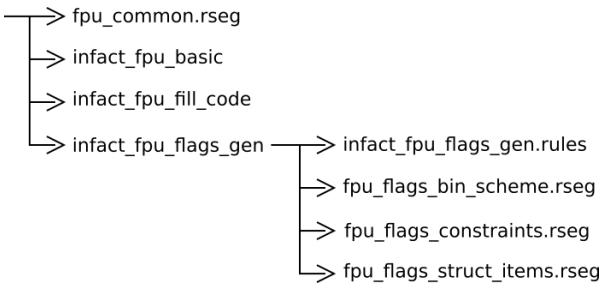
Resetovací sekvence je realizována pomocí smyčky definované klíčovým slovem `repeat` následovaným parametrem [0..7], který ponechává určitou volnost algoritmu nástroje při určení počtu taktů s resetem nastaveným na aktivní hodnotu. Smyčka je umístěna do konstrukce `symbol` s názvem `reset` v tomto souboru a díky tomu je možné ji umístit do libovolné testovací komponenty podle potřeby. Uvnitř smyčky je umístěna instance struktury `item_reset` generující transakce s aktivní hodnotou resetu. Aktivní hodnota resetu je nastavena pomoci konstrukce *attributes*, kde je hodnota meta akce `RST` nastavená skrze statický constraint uvnitř struktury na neaktivní hodnotu resetu změněna na hodnotu ak-

tivní. Definice akce, která by určovala počet resetovacích sekvencí, je provedena v testovacích komponentách, kde je umístěna, samostatně a lze tedy pro každou z nich zvolit jiný počet jejího provedení. Druhá instance struktury (`item_stuffing`) vytvořená pro vkládání resetovacích sekvencí není přímo její součástí, ale je umístěna na začátku hlavní smyčky grafu všech komponent. Slouží jako zpoždění o jeden takt nutné po provedení resetovací sekvence. Bylo také nutné nastavit pomocí konstrukce *attributes* meta akci `fpu_start` této instance na hodnotu nespouštějící operaci na FPU modulu, která je jinak nastavena pomocí statického constraintu uvnitř struktury. Instance struktury `item_stuffing` tedy zajišťuje správné navázání následujícího verifikačního scénáře.

Čekání na dokončení operace je realizováno pomocí smyčky, kde jako parametr je použito klíčové slovo `action_limited` udávající, že smyčka může být ukončena pouze pomocí metody grafu *stop_loop_expansion_after()*. Smyčka je umístěna do konstrukce `symbol` nesoucí název `wait_done`. Uvnitř smyčky je instance struktury `item_wait` generující transakce a rovněž je určena pro zaregistrování připravenosti signalizované na výstupu `ready` po dokončení operace signalizované na výstupu `done` FPU modulu. Uvnitř této smyčky je také umístěna akce `break_waiting` definována v tomto souboru, ve které je implementováno okamžité ukončení smyčky po nastavení signálu připravenosti FPU modulu na další operaci.

### 6.3.2 Verifikační scénáře

Implementace jednotlivých testovacích komponent generujících navrhované verifikační scénáře pro pokrytí různých funkcí FPU modulu během simulace, je dále rozdělena do několika segmentových souborů. Toto rozdělení je stejné pro všechny testovací komponenty a lze jej vidět na obrázku 6.13 pro testovací komponentu *infact_fpu_flags_gen*.




Obrázek 6.13: Souborová hierarchie testovacích komponent modulu FPU

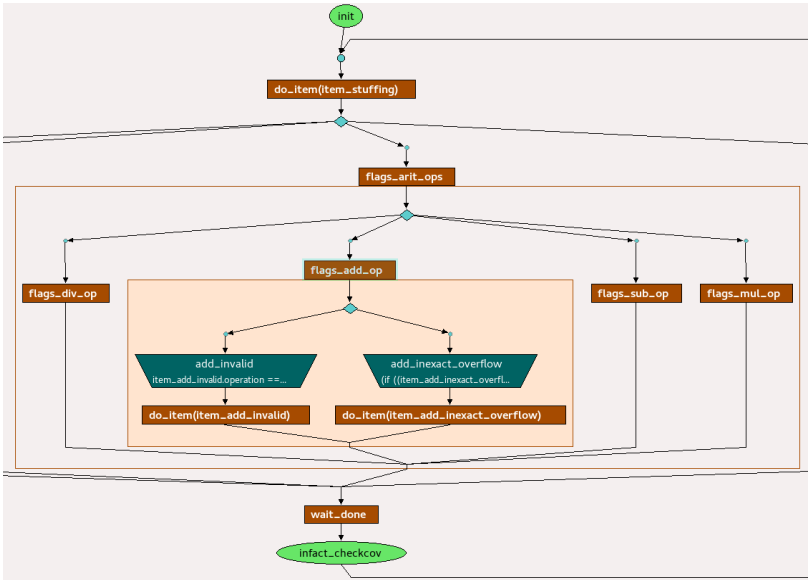
Generování verifikačních scénářů je realizováno pomocí větvení. Pro každou větev je vytvořen dynamický constraint a instance struktury, která spouští operaci definovanou v této větvi s parametry odpovídajícím zamýšlenému pokrytí generovaným verifikačním scénářem v této větvi. Z dané instance jsou vždy zahrnuty do strategie pokrytí ty meta akce, které ovlivňují danou operaci a podle potřeby je využito jak pokrytí akce (angl. *action coverage*) tak pokrytí cesty (angl. *path coverage*). Pro vymezení hodnot, které mohou být pro tento verifikační scénář generovány, je využito konstrukce `bin_scheme`, jenž je přiřazena definované strategii pokrytí nad instancí struktury v dané větvi. Pro vytvoření správného verifikačního scénáře pomocí hodnot generovaných algoritmem nástroje je tedy využito dvojího omezení, kdy nejdříve je vymezen rozsah hodnot pomocí schémat binů a následně je tento výběr ještě více zpřesněn pomocí dynamických constraintů v dané větvi. Binová schémata jsou vytvářena přímo pro konkrétní instance struktur v jednotlivých větvích. Cílem

původního návrhu bylo tato schémata vytvořit nezávisle na instanci struktury, aby bylo možné je přiřazovat k strategiím pokrytí definovaných nad libovolnou instancí struktury. Jejich implementace byla proto vložena přímo do struktury, avšak později bylo zjištěno, že je tento přístup znatelně pomalejší a to v některých měřených případech až 60 krát.

Všechny implementace verifikačních scénářů jsou koncipovány strukturně stejně a liší se pouze návrhem constraintů a binových schémat pro dosažení zamýšleného pokrytí danými verifikačními scénáři. Testovací komponenta *infact_fpu_flags_gen*, na níž je ilustrován popsaný přístup k návrhu verifikačních scénářů pro modul FPU, generuje verifikační scénáře pokrývající nastavení všech možných příznaků a jejich kombinací operacemi poskytovanými tímto modulem. Definice sekvence pravidel této testovací komponenty je na obrázku 6.14a a její grafová reprezentace je pak na obrázku 6.14b. Jak je patrné z těchto dvou obrázků, je zavedena určitá hierarchie pomocí konstrukce *symbol*, která zpřehledňuje jak definici sekvence pravidel tak i její grafovou reprezentaci.

```
infact_fpu_flags_gen = init repeat {
    do_item(item_stuffing)
    (
            flags_comp_ops
            |
            flags_convert_ops
            |
            flags_arit_ops
            |
            flags_minmax_op
    )

    wait_done

    infact_checkcov
};
```

(a) Definice kódu sekvence pravidel




(b) Grafová reprezentace sekvence pravidel

Obrázek 6.14: Testovací komponenta *infact_fpu_flags_gen*

Nyní je popsaný přístup ke generování verifikačních scénářů ilustrován na několika konkrétních příkladech, které jsou součástí této testovací komponenty. Jako první je rozebrána implementace verifikačních scénářů pro generování příznaků *inexact* a *overflow* při operaci násobení. Na obrázku 6.15, lze vidět určení rozsahů pro jednotlivé operandy násobení, ze kterých může algoritmus nástroje vybírat. Tyto rozsahy jsou rozděleny pomocí operátoru /, každý na 10 samostatných binů, z nichž ze všech algoritmus nástroje vybere hodnotu v průběhu simulace. Strategie pokrytí je v tomto případě definována pro každý operand zvlášť jako pokrytí akce a je tedy vygenerováno celkem 20 verifikačních scénářů z této větve.

Dynamický constraint ukázaný na obrázku 6.16 dále omezuje rozsahy pro generované operandy, kdy je určeno omezení pro hodnotu exponentu na rozsah 200 až 254 v jeho binární reprezentaci podle standardu IEEE 754. Toto omezení je přidáno, protože algoritmus nástroje nerozpoznává binární reprezentaci čísla s plovoucí řádovou čárkou podle standardu IEEE 754 a generuje pro něj pouze binární hodnoty odpovídající rozsahu definovaných binů. Další částí tohoto constraintu je omezení výběru módu zaokrouhlení v závislosti na znamén-

```
bin_scheme op_mul_inexact_overflow{
    item_mul_inexact_overflow.src_A
        [0x64000000..POS_MAX_NORM_VALUE]/10//big positive numbers
        [0xe4000000..NEG_MAX_NORM_VALUE]/10//big negative numbers
        ;
    item_mul_inexact_overflow.src_B
        [0x64000000..POS_MAX_NORM_VALUE]/10//big positive numbers
        [0xe4000000..NEG_MAX_NORM_VALUE]/10//big negative numbers
        ;
}
```

Obrázek 6.15: Konstrukce *bin_scheme* pro operaci násobení s příznaky

```
constraint mul_inexact_overflow dynamic{
    if(item_mul_inexact_overflow.src_A[31] == item_mul_inexact_overflow.src_B[31])
    {
        item_mul_inexact_overflow.round_mode != FPU_N_INF //this round mode doesn't allow +inf
    }
    else
    {
        item_mul_inexact_overflow.round_mode != FPU_P_INF //this round mode doesn't allow -inf
    }

    item_mul_inexact_overflow.operation == FPU_MUL &&

    item_mul_inexact_overflow.src_A[30:23] inside [200..254] &&
    item_mul_inexact_overflow.src_B[30:23] inside [200..254]
}
```

Obrázek 6.16: Dynamický constraint pro operaci násobení s příznaky

kách vybraných hodnot algoritmem nástroje, protože při některých kombinacích by nedošlo k nastavení požadovaných příznaků na výstupu po dokončení operace násobení. V případě, že jsou vybrány operandy se stejnými znaménky nemůže být vybrán mód zaokrouhlení k -∞, protože nedovoluje výsledek roven +∞ a nedošlo by tak k nastavení požadovaných příznaků *inexact* a *overflow* (případě rozdílných znamének operandů je tomu naopak). Je zde také omezení určující, že bude provedena operace násobení, ale to by mohlo být realizováno i jako součást binového schématu, záleží tedy na verifikačním inženýrovi kam toto omezení zařadí.

Další implementace verifikačního scénáře je určena pro generování příznaku *inexact* při operaci konverze čísla typu float na číslo typu integer. Pomocí konstrukce *bin_scheme* jsou určeny rozsahy, které mají největší pravděpodobnost pro vygenerování necelého čísla. Pro zvýšení této pravděpodobnosti je v dynamickém constraintu pro tuto větev, ještě přidáno omezení, že exponent nepřesáhne hodnotu 128 a bude se tedy jednat vždy o denormalizované číslo. Tady ovšem možnosti omezení za pomocí konstrukcí jazyka tohoto nástroje pro generování požadovaných hodnot končí. Pro dosažení 100% pravděpodobnosti, že dojde k vygenerování necelého čísla pro tuto operaci a dojde tedy k nastavení příznaku *inexact*, bylo nutné reimplementovat metodu meta akce `src_A`. Tato metoda nejdříve zkontroluje zda hodnota vybrána algoritmem nástroje vede k nastavení požadovaného příznaku, pokud ne, vybranou hodnotu patřičně upraví. Implementace metody je realizována přímo v hlavním souboru testovací komponenty pomocí konstrukce *attributes*. Celá implementace metody je ukázána na obrázku 6.17, kde hodnota generovaná nástrojem nese označení `meta_val`.

```
//conv_float_to_int_inexact
attributes item_conv_f2i_inexact.src_A{
    action_stmt =
    "
    const int maxValue = 2147483646;//real max value is 2147483647 but we need some space for rounding
    shortreal tmp = 0.0;
    int whole_part = 0;
    bit flag = 0;
    tmp = $bitstoshortreal(meta_val);


    if(tmp < 0) begin
        tmp *= -1;
        flag = 1;
    end


    if(tmp > maxValue) begin
        tmp = maxValue;
    end

    whole_part = tmp;

    if( tmp - whole_part == 0) begin
        tmp += 0.1;
    end

    if (flag == 1) begin
        tmp *= -1;
    end
    m_item_conv_f2i_inexact.src_A = $shortrealtobits(tmp);
    ";
}
```

Obrázek 6.17: Kód metody pro meta akci `src_A` instance `item_conv_f2i_inexact`

Následuje stručný popis zbývajících testovacích komponent. Testovací komponenta *infact_fpu_basic* je určena pro zcela náhodné testování každé operace poskytované modulem FPU a neobsahuje tedy žádné constrainty. Původním záměrem bylo využít binů definovaných uvnitř struktury, ale nakonec byly definovány schémata binů i pro tuto testovací komponentu, aby nedocházelo ke zbytečnému generování verifikačních scénářů s hodnotami, které již byly použity v případě testovací komponenty *infact_fpu_flags_gen*. Jako jediná obsahuje generování resetovací sekvence implementované v souboru *fpu_common.rseg*.

Poslední testovací komponenta byla vytvořena za účelem generování verifikačních scénářů pro dosažení co nejvyššího strukturního pokrytí, které bylo hlavní metrikou při vytváření verifikačních scénářů pro FPU modul. Bylo zde také nutné doimplementovat jednu strukturně odlišnou větev, kde po spuštění operace následují modifikované konstrukce pro čekání na připravenost FPU modulu na další operaci a pro resetovací sekvenci, kde obě smyčky obsažené v těchto konstrukcích jsou provedeny právě třikrát a instance struktur uvnitř těchto smyček nastavují signál `fpu_start`.

## 6.4 Portování

Cílem této části práce bylo přenést implementovaná pravidla definující verifikační scénáře pro moduly časovače a FPU z blokové úrovně na systémovou úroveň a navrhnout způsob jejich využití během verifikace na této úrovni. Na systémové úrovni je možné provádět společně hardwarovou i softwarovou verifikaci obvodu, kdy softwarová část je realizována programem nahraným na procesor, který je řídící jednotkou. Byl vytvořen návrh využívající implementovaná pravidla pro modul FPU ke generování verifikačních scénářů v podobě programu a pravidla pro modul časovače ke generování verifikačních scénářů v podobě sekvencí transakcí. Verifikace modulu časovače byla ponechána hardwarově pro lepší možnosti verifikace oproti softwarovému způsobu verifikace tohoto modulu.

Jak je uvedeno v podkapitole 3.2.6 jsou pravidla definovaná pomocí nástroje Questa InFact překládána do jazyka určeného verifikačním inženýrem podle potřeb verifikačního prostředí. Jednou z nabízených variant je softwarově řízená verifikace (angl. *Software Driven Verification*, SDV), která poskytuje možnost překládat definovaná pravidla nejen do jazyka verifikačního prostředí, ale také do jazyka C. Tato varianta dále poskytuje dva způsoby jakými lze řídit softwarovou část verifikace. První z nich je dynamické řízení, kdy program nahraný do procesoru komunikuje pomocí zpráv a upraveného sequenceru přímo s algoritmem nástroje řídící generování hodnot verifikačního scénáře procházením grafu během simulace. Druhou variantou je statické řízení, kdy hodnoty generované nástrojem a jejich pořadí, které určují verifikační scénáře jsou již součástí programu nahrávaného do procesoru.

Nástroj Questa InFact pouze generuje hodnoty, které jsou na blokové úrovni následně přivedeny pomocí definovaného rozhraní na vstupy verifikovaného hardwaru. Programová paralela k těmto vstupům, na něž se přivádí generované hodnoty, je funkce s odpovídajícím počtem parametrů, vyvolávající odpovídající operace s nimi. Programy pro procesor Berkelium od společnosti Codasip jsou psány v jazyce C, pro který je nástroj schopen provést překlad definovaných pravidel. Byla vytvořena funkce `fpu_op` umístěna do samostatného souboru `fpu_functions.c` s parametry pro vstupní operandy, zvolenou operaci, suboperaci a mód zaokrouhlení. Ve struktuře představující transakci byla dále definována akce `body`, ve které byla vytvořená funkce volána s odpovídajícími hodnotami generovanými nástrojem Questa InFact (viz obrázek 6.18). Klíčové slovo `${item}` umožňuje definici volání funkce umístit přímo do struktury, kdy je následně součástí každé vytvořené instance této struktury.

```
action body;

attributes body{
    sw_action_stmt =
    "
    fpu_op(
    ${item}.src_A,
    ${item}.src_B,
    ${item}.operation,
    ${item}.sub_op,
    ${item}.round_mode);
    "
}
```

Obrázek 6.18: Kód metody pro akci `body`

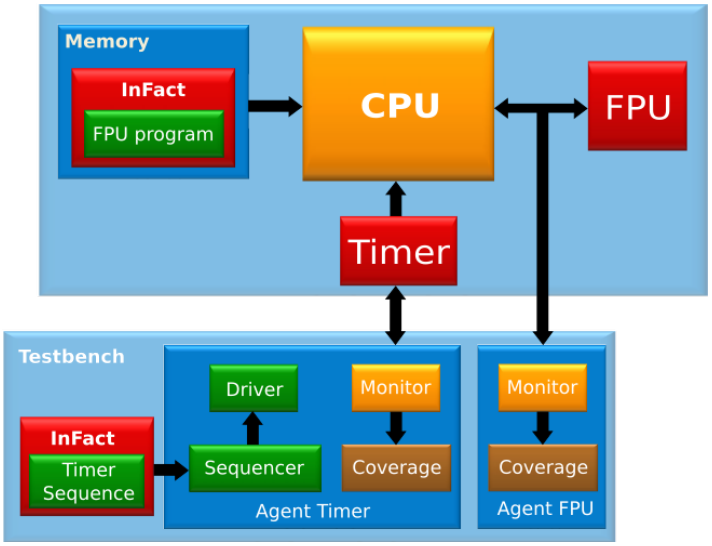
Ne všechny verifikované vlastnosti FPU modulu bylo možné ověřit pomocí jazyka C, jako příklad lze uvést možnost určení módu zaokrouhlovaní při provádění některých z operací na modulu FPU a bylo proto rovněž také potřeba využít jazyka *assembler* (Jazyk Symbolických Instrukcí, JSA) určujícím konkrétní instrukce používající generované hodnoty. Ukázka části kódu implementované funkce `fpu_op` je vidět na obrázku 6.19, který zodpovídá za vyvolání operace násobení se zaokrouhlením směrem k $+\infty$.

```
case FPU_P_INF:
    __asm__ __volatile__(
        "fadd.s (up) f0 , f0 , f1"
    );
    break;
```

Obrázek 6.19: Kód assembleru pro násobení

V této práci byla dána přednost statické možnosti řízení softwarové části verifikace a tedy většímu programu nahrávanému do procesoru, před případným negativním dopadem

komunikace na dobu běhu simulace při využití dynamické varianty. Díky tomu, že je verifikace FPU modulu prováděna na systémové úrovni pomocí programu a absenci úpravy sequenceru díky využití statického řízení generovaného programu, není třeba provádět žádné úpravy verifikačního prostředí pro tuto část návrhu. Pro možnost přímého řízení modulu časovače na systémové úrovni bylo třeba provést úpravy poskytnutého verifikačního prostředí, ve kterém byl již modul připojen k procesoru. Pro minimalizaci zásahu do verifikačního prostředí byl vytvořen nový agent pro modul časovače, zprostředkovávající sekvence transakcí během simulace, zatímco stávající agent byl ponechán pro již zapojené sledování hodnot na vstupech/výstupech modulu pro měření funkčního pokrytí. Blokový diagram výsledného zapojení lze vidět na obrázku 6.20.




Obrázek 6.20: Blokové zapojení návrhu na systémové úrovni

Z testovacích komponent, obsahující pravidla implementovaná pro modul FPU, bylo možné odstranit rozhraní zprostředkovávající komunikaci s vstupními a výstupními signály modulů a také několik nepotřebných meta akcí a s nimi spojených konstrukcí, jejichž přítomnost by vyvolala jejich zbytečné generování a tedy další zvětšení a zpomalení programu. Příkladem takových signálů jsou `fpu_start` zahajující operaci na FPU modulu, či výstupní signál `ready` využívaný při verifikaci na blokové úrovni k signalizaci, že je možné přejít na další verifikační scénář.

Každá z testovacích komponent pro modul FPU byla převedena do samostatného programu nahrávaného do procesoru. Verifikační scénáře implementované pro modul časovače byly využity pro plnou verifikaci modulu na začátku simulace. Verifikace FPU pomocí verifikačních scénářů v podobě programů trvá mnohem déle, než je tomu na blokové úrovni při přímém ovlivňování vstupů a proto byla vytvořena nová testovací komponenta pro modul časovače obsahující pouze verifikační scénáře pro ověřování jednotlivých módů upravené tak, aby doba mezi přerušeními odpovídala poměrově průměrnému simulačnímu času. Průměrný simulační čas pro dokončení programů vytvořených z jednotlivých testovacích komponent pro modul FPU bez přerušení byl 5 000 000 ns, s dobou jednoho hodinového taktu rovnu 10 ns, byla určena mezera mezi přerušeními 25 000 až 50 000 hodinových taktů, kdy volba z tohoto rozsahu je ponechána na algoritmu nástroje, přičemž každý z těchto módů vyvolá přerušení právě třikrát.

# Kapitola 7

# Výsledky a jejich analýza

V této kapitole je postupně provedena analýza dosažených výsledků u jednotlivých modulů a také možností portovatelnosti implementovaných verifikačních scénářů pro jejich využití při verifikaci na systémové úrovni. Dále je zde popsáno několik experimentů, které byly prováděny za účelem prozkoumání možností nástroje Questa InFact postaveném na portovatelných stimulech využívaném v této práci.

## 7.1 Analýza výsledků

Tato podkapitola se věnuje postupně analýze dosažených výsledků u jednotlivých modulů a také možností portovatelnosti implementovaných verifikačních scénářů pro jejich využití při verifikaci na systémové úrovni

### 7.1.1 Časovač

V případě modulu časovače je pro analýzu dosažených výsledků využito funkční pokrytí definované ve verifikačním prostředí, jenž bylo také hlavní metrikou pří tvorbě pravidel tvořící verifikační scénáře pro tento modul. Definované funkční pokrytí v sobě zahrnuje 172 různých vlastností, které je potřeba v průběhu verifikace modulu časovače ověřit. V ideálním případě by mělo dojít k ověření každé z definovaných vlastností právě jednou, tedy pokud není verifikačním inženýrem určeno jinak. Tohoto však dosáhnout nelze, neboť například pro datový vstup s rozsahem hodnot 0 až $2^{32} - 1$ je v případě tohoto modulu definováno 64 binů, a je tedy potřeba minimálního počtu 64 transakcí, které budou obsahovat vždy hodnotu z jiného rozsahu určeného jedním z těchto binů, ale zároveň například pro 1 bitový resetovací vstup lze definovat pouze dva biny, tedy postačí 2 transakce pro jeho pokrytí. Z daného vyplývá, že není možné se redundanci úplně vyhnout a je tedy hlavní snahou, co nejvíce snížit počet transakcí pro pokrytí všech definovaných vlastností. Pro potřeby analýzy byl zaveden vzorec pro výpočet poměru potřebných transakcí pro plné pokrytí bez a za použití nástroje Questa InFact:

$$\frac{\text{počet transakcí}}{\text{počet transakcí s QI}} = \text{poměr počtu transakcí} \tag{7.1}$$

V původní verzi verifikace, bez využití nástroje Questa InFact pro řízení generování stimulů, bylo dosaženo při 50 000 transakcích 98.1% funkčního pokrytí, pro dosažení 100%

funkčního pokrytí modulu časovače bylo potřeba až 75 000 transakcí. Lze tedy vidět, že bez řízení je nutných až 25 000 dalších transakcí pro pokrytí zbývajících 1.9%. Tento vysoký počet transakcí, ať už pro pokrytí celkové nebo chybějících 1.9%, je dán značnou redundancí při jejich generování. S využitím nástroje Questa InFact založeném na portovatelných stimulech bylo potřeba vygenerovat 979 transakcí pro dosažení 100% funkčního pokrytí. Po dosazení do vzorce 7.1 dává tento výsledek **76** násobné zlepšení oproti verifikaci provedené bez využití tohoto nástroje (viz také první řádek tabulky 7.1). V tabulce je uvedeno, že pro tento *seed* nebylo dosaženo 100% strukturního pokrytí. Původně byl návrh pravidel prováděn v jednom souboru jako jeden celek a bylo dosaženo plného pokrytí. Tento pokles je dán tím, že při jakékoliv změně v pravidlech popisující verifikační scénáře se změní rovněž vyhodnocování grafu nástrojem a to i při jeho rozdělení pomocí hierarchie. Tento aspekt nástroje byl také mimochodem uveden jako hlavní důvod rozdělení testovací komponenty pro modul FPU na několik menších.

Tabulka 7.1: Naměřené hodnoty grafu verze 1 pro modul časovače

| Seed | Počet transakcí | Pokrytí | |
|---|---|---|---|
| | | Strukturní | Funkční |
| 746258445 | 979 | 97,22 | 100 |
| 428476545 | 948 | 100 | 100 |
| 228558778 | 944 | 100 | 100 |
| 131313131 | 906 | 100 | 100 |
| 992641357 | 975 | 97,22 | 100 |

Graf (viz 6.8b), který vedl k těmto výsledkům, je sestaven tak, aby byl ponechán určitý prostor pro pseudonáhodné generování transakcí ovlivněné pouze omezeními definovanými uvnitř transakce samotné. Jako příklad lze uvést generování čtecích a zápisových požadavků s vyšší pravděpodobností oproti ostatním. Na obrázku 7.1 je pak ukázáno dosažené pokrytí tak, jak jej zobrazuje simulační nástroj QuestaSim, ze kterého lze vyčíst, že během simulace bylo skutečně vygenerováno mnohem více transakcí zahrnujících požadavky na čtení a zápis.


Obrázek 7.1: Využití pravděpodobnosti ke generování více read/write požadavků

### 7.1.2 FPU

V případě FPU modulu je pro analýzu dosažených výsledků využito strukturní pokrytí, automaticky vygenerované použitým simulátorem QuestaSim, které bylo rovněž hlavní metrikou při tvorbě pravidel tvořící verifikační scénáře pro tento modul. Plné pokrytí je pro

FPU modul stanoveno na 96.2%, neboť verifikovaný modul obsahuje také 64 bitovou verzi operací konverze datových typů, kterou nelze s poskytnutým verifikačním prostředím, vytvořeným pro 32 bitovou verzi FPU modulu, verifikovat a nebylo tedy možné dosáhnout 100% strukturního pokrytí. Pro dosažení tohoto strukturního pokrytí bylo bez využití portovatelných stimulů potřeba 50 000 až 100 000 transakcí. S využitím nástroje Questa InFact bylo potřeba vygenerovat nejvíce 2057 transakcí (viz první řádek tabulky 7.2) pro dosažení stanoveného strukturního pokrytí. Po dosazení do vzorce 7.1 dává tento nejhorší výsledek **24** násobné zlepšení oproti nejlepšímu výsledku 50 000 transakcí, dosaženým bez využití nástroje Questa InFact pro řízení jejich generování.

V tabulce lze vidět, že není dosaženo 100% funkčního pokrytí, které bylo definováno až po implementaci verifikačních scénářů pro FPU modul. Chybějící pokrytí odpovídá výstupu, na který je přiváděn výsledek operace. Jeho plné pokrytí by mohlo být realizováno například pomocí verifikačního scénáře tvořícího operaci sčítání, kdy jeden z operandů by byl roven nule a druhý by odpovídal postupně všem hodnotám požadovaným na výstupu, ale tvorba takového verifikačního scénáře se jeví zbytečná. Funkční pokrytí bylo využito během experimentování s nástrojem a vytvořenými verifikačními scénáři pro FPU modul.

Tabulka 7.2: Naměřené hodnoty grafu verze 1 pro FPU modul

| **Seed** | **Počet transakcí** | **Pokrytí** | |
|---|---|---|---|
| | | **Strukturní** | **Funkční** |
| 252964 | 2057 | 96,2 | 96,39 |
| 746913 | 1800 | 96,2 | 97,02 |
| 583091 | 1849 | 96,2 | 96,39 |

### 7.1.3 Portování

Pro vyhodnocení možnosti portování byla analyzována hlavně časová náročnost simulace, která je největší překážkou pro zavedení pseudonáhodných verifikačních testů na systémové úrovni. V tabulce 7.3 lze vidět simulační časy jednotlivých programů vygenerovaných pomocí testovacích komponent pro modul FPU běžících během simulace zároveň s verifikačními scénáři přiváděnými přímo na vstupy modulu časovače. Simulační čas se zvětšil oproti průměrné hodnotě 5 000 000 ns, protože bylo přidáno generování přerušení modulem časovače, které je následně přiváděno do procesoru v průběhu provádění programu nahraného pro verifikaci FPU modulu. Simulační čas je na této úrovni o mnoho vyšší oproti blokové úrovni, kde celkový simulační čas běhu všech verifikačních scénářů pro modul FPU je 162 920 ns. Je to dáno komplexním chováním na systémové úrovni, kdy je například nejdříve potřeba zpracovat jednotlivé instrukce nahraného programu na rozdíl od blokové úrovně, kde jsou hodnoty přiváděny přímo na vstupy FPU modulu a může tak dojít k okamžitému zpracování.

Pro zjištění teoretického zlepšení při využití portovatelných stimulů lze vzít do úvahy, že v nejhorším případě, tedy po dosazení počtu transakcí nejhoršího výsledku za použití nástroje Questa InFact a nejlepšího výsledku bez jeho použití do vzorce 7.1, je počet transakcí 24 krát menší při běhu verifikačních scénářů generovaných ze všech testovacích komponent

pro FPU modul. Pro výpočet teoretické doby trvání verifikace pomocí programů byl zaveden tento vzorec:

$$\sum_{i=0}^{n} \text{sim. čas prog. s QI[i]} * \frac{\text{počet transakcí}}{\text{počet transakcí s QI}} = \text{celkový sim. čas prog.}, \tag{7.2}$$

kde je nejdříve proveden součet simulačních časů všech programů představujících jednotlivé testovací komponenty a následně je tato hodnota vynásobena poměrem počtu transakcí potřebných pro plné pokrytí bez a za použití nástroje Questa InFact. Celkový simulační čas verifikačních scénářů ve formě programů je roven 24 262 650 ns a lze předpokládat, že bez využití nástroje Questa InFact by byl tento čas nejméně 24 krát horší.

Tabulka 7.3: Simulační časy společné hardwarové a softwarové verifikace na systémové úrovni

| **Program** | **Simulační čas [ns]** |
|---|---|
| fpu_basic.c | 7 739 570 |
| fpu_flags_gen.c | 8 261 540 |
| fpu_fill_code.c | 8 261 540 |

## 7.2 Další experimenty a úvahy

Pro implementované verifikační scénáře modulu časovače, byla provedena změna struktury grafu. Jednotlivé podgrafy ověřující různé vlastnosti byly přesunuty do samostatných větví podobně jako je tomu v případě implementace pro FPU modul. Touto transformací struktury byl snížen počet potřebných transakcí až na 491, což po dosazení do vzorce 7.1 dává ještě větší **152** násobné zlepšení oproti verifikaci bez využití nástroje Questa InFact (viz první řádek tabulky 7.4) a dokonce bylo v tomto případě dosaženo i 100% strukturního pokrytí.

Na rozdíl od předchozího uspořádání nedává tento graf příliš prostoru pro pseudonáhodné generování transakcí. Stojí za zvážení zda přílišné snížení náhodnosti nemůže funkční verifikaci spíše uškodit, protože by nemusely být odhaleny chyby, na které nebylo myšleno při specifikaci, ani při tvorbě verifikačního plánu.

Další experiment byl proveden na verifikačních scénářích implementovaných pro FPU modul. Verifikace využívající portovatelných stimulů je značně zkrácena díky řízenému generování. Tento uspořený čas lze využít k dalšímu testování a proto byla změněna strategie pokrytí testovací komponenty *infact_fpu_basic* z pokrytí akcí na pokrytí cest, což značně navýšilo počet vygenerovaných transakcí (viz první řádek tabulky 7.5). Tato změna měla do určité míry nahradit čistě pseudonáhodné generování transakcí, při kterém je určen pouze požadovaný počet transakcí a může tedy docházet k značné redundanci a ztratit tak účinek. Stojí za zvážení, zda verifikace všech možných kombinací je vhodnou náhradou za redundantní pseudonáhodné generování transakcí, neboť oba způsoby mohou odhalit stejnou chybu, ale různou rychlostí. Pomocí provedené změny bylo dosaženo 99,84% funkčního pokrytí a také byla odhalena další chyba v návrhu.

Tabulka 7.4: Naměřené hodnoty grafu verze 2 pro modul časovače

| Seed | Počet transakcí | Pokrytí | |
|---|---|---|---|
| | | Strukturní | Funkční |
| 746258445 | 491 | 100 | 100 |
| 428476545 | 561 | 100 | 100 |
| 228558778 | 498 | 100 | 100 |
| 131313131 | 581 | 100 | 100 |
| 992641357 | 578 | 100 | 100 |

Algoritmus nástroje Questa InFact generuje verifikační scénáře s využitím pseudonáhodného generování hodnot pro jednotlivé stimuly obsažené v transakcích. Pro ověření míry vlivu pseudonáhodného generování hodnot na implementovaných verifikačních scénářích, byly měněny *seedy* pro jejich generování. Tyto změny měly vliv na výsledný počet transakcí a v některých případech i na dosažené pokrytí a počet odhalených chyb. Výsledky měření s různými seedy je možné vidět v tabulkách 7.1 až 7.5.

Tabulka 7.5: Naměřené hodnoty grafu verze 2 pro FPU modul

| Seed | Počet transakcí | Pokrytí | |
|---|---|---|---|
| | | Strukturní | Funkční |
| 252964 | 6976 | 96,21 | 99,84 |
| 746913 | 6911 | 96,22 | 100 |
| 583091 | 6665 | 96,2 | 100 |

Pokud je nalezen *seed*, pro který nedošlo k 100% pokrytí, je na verifikačním inženýrovi zda doimplementuje pravidla, která by vedla k vygenerování dalších verifikačních scénářů pokrývající objevené mezery v pokrytí nebo zda je dostačující, že je úplného pokrytí dosaženo pomocí jiného *seedu*, který však nemusel být doposud použit a bylo by jej potřeba teprve nalézt. Je nutné vzít také do úvahy vliv doimplementovaných pravidel, která změní průběh generování verifikačních scénářů a mohla by tak vytvořit další mezery v pokrytí, které byly doposud pokryty pomocí pseudonáhodného aspektu řízeného generování.

Při nastavení jednoho ze *seedů* v druhé testované verzi grafu pro modul časovače došlo také k tomu, že nebyla odhalena chyba navzdory tomu, že se jednalo o případ generující druhý nejvyšší počet transakcí (viz poslední řádek tabulky 7.4). Tato skutečnost podporuje myšlenku, že by mělo být při implementaci verifikačních scénářů myšleno i na ponechání určitého prostoru pro náhodné generování transakcí, a že je rovněž také vhodné po dokončení implementace několikrát změnit *seed*. To vše by mělo značně omezit možnost neodhalení chyby v návrhu při jeho verifikaci.

# Kapitola 8

# Závěr

Neustále narůstající komplexnost hardwarových systémů ztěžuje jejich verifikaci, která se stala nedílnou součástí jejich vývoje. Verifikaci těchto systémů je čím dál složitější provádět efektivně, narůstá časová náročnost jak přípravy odpovídající verifikace pro daný systém, tak doba jejího trvání. Toto je hlavním důvodem vývoje stále dokonalejších technik pro zefektivnění verifikace.

Tato práce je zaměřená na prozkoumání možností portovatelných stimulů, jenž je jedna z těchto nově vyvíjených technik. Portovatelné stimuly mají za cíl výrazně snížit dobu verifikace skrze řízené generování stimulů, díky čemuž značně klesne jejich redundance, a skrze přenositelnost implementovaných verifikačních scénářů napříč úrovněmi abstrakce verifikovaného systému.

V této práci byly implementovány verifikační scénáře pro vybrané moduly procesoru Berkelium implementujícím architekturu RISC-V od společnosti Codasip a následně byly tyto verifikační scénáře využity pro společnou verifikaci hardwarové a softwarové části na systémové úrovni. Verifikační scénáře pro FPU modul byly použity na softwarovou část, kde pomocí nich byly volány jednotlivé operace jako příkazy v programu nahraném do procesoru a verifikační scénáře pro modul časovače pak byly použity na hardwarovou část, pro generování přerušení napříč simulací s různým odstupem.

Následně bylo při analýze dosažených výsledků zjištěno až **152** násobné snížení počtu transakcí pro dosažení plného pokrytí s využitím portovatelných stimulů při verifikaci na blokové úrovni. Byla také vyhodnocena časová náročnost prováděných verifikačních scénářů přenesených do programové podoby, kdy samotné převedení těchto scénářů bylo nenáročné a z pohledu implementovaných verifikačních scénářů došlo pouze k malým změnám (byly odstraněny (na této úrovni) zbytečné části) a to pouze za účelem optimalizace rychlosti výsledných programů. Bylo provedeno také několik experimentů na implementovaných verifikačních scénářích pro vyhodnocení několika faktorů ovlivňujících jejich generování.

Pro možnost využít pseudonáhodné generování stimulů na systémové úrovni je pro zachování efektivity řízené generování stimulů nutností, zvláště pak při neustále narůstající komplexnosti hardwarových systémů. Definice verifikačních scénářů pomocí pravidel definovaných jazykem nezávislým na jazyce verifikačního prostředí za účelem řízení generování stimulů a možnost následného přenosu těchto pravidel, ať už na jinou úroveň abstrakce nebo do jiného verifikačního prostředí, je dalším krokem vpřed k efektivnější funkční verifikaci.

# Literatura


[1] *IEEE Standard for Binary Floating-Point Arithmetic*. IEEE, 1985, ISBN 0-7381-1165-1.
URL http://ieeexplore.ieee.org/document/30711/

[2] Accellera Portable Stimulus Working Group: *PSS Early Adopter(EA) Portable Test and Stimulus Standard*. Accellera, Červen 2017.
URL http://www.accellera.org/images/downloads/drafts-review/PSS_Early_Adopter_Release.pdf

[3] Anderson, T. L.: *Verifying SoCs from the Inside Out*. Chip Design Magazine, Srpen 2012.
URL http://chipdesignmag.com/display.php?articleId=5153

[4] Ballance, M.: *Automating Tests with Portable Stimulus from IP to SoC Level* . Mentor, Březen 2017.
URL https://verificationacademy.com/verification-horizons/march-2017-volume-13-issue-1/automating-tests-with-portable-stimulus-from-ip-to-soc-level

[5] Ballance, M.: *Smoothing the Path to Software-Driven Verification with Portable Stimulus* . Mentor, Červen 2017.
URL https://verificationacademy.com/verification-horizons/june-2017-volume-13-issue-2/smoothing-the-path-to-software-driven-verification-with-portable-stimulus

[6] Codasip: *Codix Berkelium Bk1 Datasheet, Version 1.1.0, Document Version 1.2.0*. Nov 2017.

[7] Codasip: *Codix Berkelium Bk3 Datasheet, Version 1.6.0, Document Version 1.2.0*. Oct 2017.

[8] Codasip: *Codix Berkelium RISC-V Compliant Processor IP*. 2017.

[9] Codasip: *Interní specifikace FPU*. 2017.

[10] Codasip: *Interní specifikace časovače*. 2017.

[11] Fitzpatrick, T.: *UVM basics - Connecting Components*. Verification Academy, 2013.
URL https://verificationacademy.com/sessions/uvm-connecting-components

[12] Fitzpatrick, T.: *UVM basics - Connecting Env to DUT*. Verification Academy, 2013.
URL https://verificationacademy.com/sessions/uvm-connecting-env-dut

[13] Fitzpatrick, T.: *UVM basics - Introducing transactions*. Verification Academy, 2013.
URL
https://verificationacademy.com/sessions/uvm-introducing-transactions

[14] Fitzpatrick, T.: *UVM basics - Introduction to UVM*. Verification Academy, 2013.
URL https://verificationacademy.com/sessions/introduction-uvm

[15] Fitzpatrick, T.: *UVM basics - Monitors and Subscribers*. Verification Academy, 2013.
URL
https://verificationacademy.com/sessions/uvm-monitors-and-subscribers

[16] Fitzpatrick, T.: *UVM basics - UVM "Hello World"*. Verification Academy, 2013.
URL https://verificationacademy.com/sessions/uvm-hello-world/

[17] Fitzpatrick, T.: *Portable Stimulus Basics - Why Portable Stimulus* . Mentor, 2017.
URL https://verificationacademy.com/sessions/why-portable-stimulus

[18] Flake, P.: *Why SystemVerilog?* IEEE, 2008, ISBN 978-2-9530504-8-6.
URL http://ieeexplore.ieee.org/document/6646660/

[19] Lee, Y.; Schmidt, C.; Ou, A.; aj.: *The Hwacha Vector-Fetch Architecture Manual, Version 3.8.1. Technická Zpráva UCB/EECS-2015-262, EECS Department, University of California, Berkeley, Dec 2015.*
*URL*
*http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-262.html*

*[20] Mentor: Questa inFact User's Manual. Mentor, Srpen 2017.*

*[21] Mentor Verification Methodology Team: UVM Cookbook. Verification Academy, 2013.*
*URL https://verificationacademy.com/cookbook/uvm*

*[22] Nair, M.; Kumar, S.; Sairam, P.; aj.: Bridging UVM to the Portable Stimulus Standard with Questa® inFact . Mentor, Březen 2017.*
*URL https://verificationacademy.com/verification-horizons/march-2017-volume-13-issue-1/bridging-uvm-to-the-portable-stimulus-standard-with-questa-infact*

*[23] Spear, C.: SystemVerilog for verification : A Guide to Learning the Testbench Language Features. New York : Springer, 2008, ISBN 978-0-387-76529-7.*

*[24] Waterman, A.; Lee, Y.; Avizienis, R.; aj.: The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10. Technická zpráva, CS Divison, EECS Department, University of California, Berkeley, May 2017.*
*URL https://riscv.org/specifications/privileged-isa/*

*[25] Waterman, A.; Lee, Y.; Patterson, D. A.; aj.: The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2. Technická zpráva, CS Divison, EECS Department, University of California, Berkeley, May 2017.*
*URL https://riscv.org/specifications/*

*[26] Wile, B.; Goss, J.; Roesner, W.: Comprehensive Functional Verification: The Complete Industry Cycle. Elsevier Science, May 2005.*
*URL https://books.google.cz/books?id=btl_OX3kJ7MC*